ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτοϊούμενη, Ελαστική Αποθήκευση Αμετάβλητων Αντικειμένων με Υψηλή Διαθεσιμότητα ως Πρωτογενές Χαρακτηριστικό στο Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Α. Κατσακιώρης

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Οκτώβριος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτοϊούμενη, Ελαστική Αποθήκευση Αμετάβλητων Αντικειμένων με Υψηλή Διαθεσιμότητα ως Πρωτογενές Χαρακτηριστικό στο Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Α. Κατσακιώρης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Νοεμβρίου 2018.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Οκτώβριος 2018

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Highly-Available, Self-Healing, Elastic, Immutable Object Storage as a Kubernetes Primitive

DIPLOMA THESIS

Christos A. Katsakioris

Computing Systems Laboratory

Athens, October 2018

. . . . . .

Χρήστος Α. Κατσακιώρης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Abstract

The advent of the era of the Cloud and the widespread adoption of the distributed computing paradigm has lately enabled systems and applications to reach higher standards with respect to their scalability, as well as their performance overall. Containers and operating-system-level virtualization increasingly gain ground among various deployment methods across datacenters as an efficient alternative to hypervisor-based virtualization. Due to both this and the consequent prevalence of the Microservices Architecture in the landscape of application and system design, a rich ecosystem has been rapidly developed to support containerization, and still flourishes, with Kubernetes being one of its most successful representatives.

Meanwhile, the significance of data produced, either directly or indirectly, by users and collected by enterprises, is thriving. Emerging technologies revolving around Big Data and their process and analysis are inextricably linked with Cloud computing, and Cloud storage in particular. Driven by the recent commoditization of the hardware infrastructure within datacenters, distributed storage systems are employed to gradually supersede traditional storage solutions, such as SANs and proprietary hardware appliances. Notwithstanding their momentum, most of the distributed storage systems that have been widely adopted across the industry thanks to the reliability and robustness they provide, such as Ceph and Cassandra, have yet to be truly integrated with containerization utilities and platforms, like Docker and Kubernetes, due to their inevitably stateful nature.

In this thesis we attempt to tackle the hindrance of statefulness, which unavoidably ails all containerized deployments of distributed storage systems. Rather than working with existing systems, we propose the design of two new distributed object stores atop Kubernetes, along with their respective implementations, which are based on a set of intelligent, stateful agents and a fleet of stateless proxy servers. Even though their functionality is rudimentary so far, since Update operations on data are not supported, they end up undergirding remarkable attributes, including incremental scalability and elasticity, load balancing, self-healingness, and high availability and fault tolerance via data replication. These are accomplished by leveraging Kubernetes' extensive set of features, as well as various concepts, techniques and algorithms, such as immutability, content-addressability, sharding, and consistent hashing. The architecture of one of the systems involves an Operator in order to ensure the automation in deployment and management of the system, in coordination with the rest of the agents, using the controller pattern on which all components fundamentally rely.

Keywords

object store, distributed storage, immutability, content-addressability, sharding, scalability, elasticity, consistent hashing, stateful, containers, Docker, Kubernetes, controller, Operator, gRPC, rsync

Περίληψη

Η έλευση της εποχής των υπηρεσιών Cloud και η ευρεία υιοθέτηση του υποδείγματος κατανεμημένης υπολογιστικής έχει προσφάτως παράσχει τη δυνατότητα σε εφαρμογές να θέσουν τον πήχη υψηλότερα αναφορικά με την κλιμακωσιμότητά τους, αλλά και τις επιδόσεις τους γενικότερα. Τα "containers" και η εικονικοποίηση σε επίπεδο λειτουργικού συστήματος διαρκώς κερδίζουν έδαφος μεταξύ των διάφορων μεθόδων "deployment" εφαρμογών στα κέντρα δεδομένων, ως μία αποδοτική εναλλακτική λύση έναντι της εικονικοποίησης βάσει υπερεπόπτη. Χάριν τόσο του ανωτέρω, όσο και της συνεπαγόμενης καθιέρωσης της Αρχιτεκτονικής Μικροϋπηρεσιών στο χώρο του σχεδιασμού εφαρμογών και συστημάτων, ένα πλούσιο οικοσύστημα αναπτύχθηκε ταχέως, το οποίο αποσκοπεί στην υποστήριξη της τάσης για "containerization", και το οποίο μεσουρανεί, έχοντας το Kubernetes ως έναν από τους χαρακτηριστικότερους και πιο επιτυχημένους εκπροσώπους του.
Εντωμεταξύ, η χρηστικότητα των δεδομένων, των εμμέσως ή αμέσως παραγόμενων από χρήστες και περισυλλεγόμενων από εταιρείες, ανθεί ολοένα και περισσότερο. Ανερχόμενες τεχνολογίες, σχετικές με Δεδομένα Μεγάλου Όγκου (Big Data), καθώς και την επεξεργασία και ανάλυσή τους, είναι άρρηκτα συνδεδεμένες με υπολογιστικά περιβάλλοντα Cloud, ιδίως όσον αφορά την αποθήκευσή τους. Λόγω της ευρείας εμπορευματοποίησης των υποδομών υλικού των σύγχρονων κέντρων δεδομένων, οι περιπτώσεις χρήσης κατανεμημένων συστημάτων αποθήκευσης με σκοπό την σταδιακή αντικατάσταση των παραδοσιακών εξοπλισμών και συστημάτων αποθήκευσης, όπως είναι τα SAN, πληθαίνουν. Όμως, παρά την τάση αυτή, η λειτουργία των περισσότερων ευρέως διαδεδομένων και αξιόπιστων κατανεμημένων συστημάτων αποθήκευσης, όπως τα Ceph και Cassandra, μέχρι στιγμής δεν έχει ενσωματωθεί πλήρως στις αυτοματοποιημένες διαδικασίες που παρέχει η χρήση εργαλείων "containerization", όπως τα Docker και Kubernetes, εξαιτίας της ανάγκης τους να διατηρούν μόνιμη κατάσταση (statefulness).
Το παρόν πόνημα αποτελεί μια προσπάθεια ικανοποίησης της ανάγκης για διατήρηση μόνιμης κατάστασης, που αναπόδραστα επηρεάζει κάθε "containerized deployment" κατανεμημένου συστήματος αποθήκευσης. Παρουσιάζεται ο σχεδιασμός δύο νέων κατανεμημένων συστημάτων αποθήκευσης αντικειμένων (distributed object stores) με βάση το Kubernetes, καθώς και οι αντίστοιχες υλοποιήσεις τους, τα οποία βασίζονται σε ένα σύνολο ευφυών "agents" και "proxy servers". Μολονότι, προς το παρόν, τα συστήματα περιορίζονται σε στοιχειώδεις λειτουργίες, ιδίως λαμβάνοντας υπ᾽όψιν ότι δεν υποστηρίζονται μεταβολές σε ήδη αποθηκευμένα δεδομένα, εξασφαλίζονται αξιοσημείωτες ιδιότητες, μεταξύ άλλων: κλιμακωσιμότητα και ελαστικότητα, ισοκατανομή φόρτου (load balancing), αυτοΐαση (self-healingness), αλλά και υψηλή διαθεσιμότητα (high availability) και ανοχή σε σφάλματα (fault tolerance) μέσω αυτοματοποιημένης δημιουργίας αντιγράφων των δεδομένων (data replication). Αυτά επιτυγχάνονται χάρη στο ευρύ φάσμα δυνατοτήτων που παρέχεται από το Kubernetes, καθώς και σε ποικίλες έννοιες, τεχνικές και αλγορίθμους που χρησιμοποιούνται, όπως η αμεταβλητότητα (immutability), η διευθυνσιοδότηση-βάσει-περιεχομένου (content-addressability), ο τεμαχισμός (sharding) και ο αλγόριθμος συνεπούς κατακερματισμού (consistent hashing). Η δε αρχιτεκτονική ενός εκ των δύο συστημάτων, περιλαμβάνει έναν "Operator" ώστε, σε συνεργασία με τους "agents", και με τη συνδρομή του προτύπου ελεγκτή που εφαρμόζεται από όλα τα στοιχεία του συστήματος, να διασφαλιστεί η αυτοματοποιημένη διαχείριση κάποιων σύνθετων πτυχών του.

Λέξεις-Κλειδιά

object store, distributed storage, immutability, content-addressability, sharding, scalability, elasticity, consistent hashing, stateful, containers, Docker, Kubernetes, controller, Operator, gRPC, rsync

Preface

Οφείλω να ευχαριστήσω στο σημείο αυτό, τον επιβλέποντα της διπλωματικής μου, Καθηγητή Νεκτάριο Κοζύρη, ο οποίος πρώτος μου κοινώνησε το ενδιαφέρον του για τον κόσμο τον υπολογιστικών συστημάτων μέσω των διαλέξεών του.

Θα ήθελα να εκφράσω την ιδιαίτερη ευγνωμοσύνη μου προς τον Διδάκτορα Βαγγέλη Κούκη για την αδιάλειπτη και καθοριστική συμβολή του, τόσο στην καλλιέργεια του ενδιαφέροντός μου για ποικίλες πτυχές των υπολογιστικών συστημάτων, όσο και στην διαμόρφωση του τρόπου σκέψης μου ως προς την προσέγγιση σχετικών ζητημάτων. Θέλω να ευχαριστήσω επίσης τον Δημήτρη Αραγιώργη, για τον χρόνο που μου αφιέρωσε και την καίρια βοήθεια που μου παρείχε σε τεχνικά ζητήματα της διπλωματικής μου, τόσο άμεσα όσο και έμμεσα, θέτοντάς μου τα κατάλληλα ερωτήματα ώστε να επανατοποθετηθώ στη σωστή κατεύθυνση για να συνεχίσω.

Οφείλω ακόμα ένα μεγάλο ευχαριστώ στον αδελφό μου (και συνάδελφό μου) Ηλία, ο οποίος μοιράστηκε μαζί μου τις σκέψεις του, αλλά και άκουσε τις δικές μου, αναφορικά με διάφορα θέματα που ανέκυψαν κατά τη διάρκεια της εκπόνησης της παρούσας εργασίας, καθώς και στη συνάδελφό μου Γεωργία Κοκκίνου για τη συχνή και χρήσιμη ανταλλαγή απόψεων γύρω από το κοινό πεδίο του ενδιαφέροντός μας.

Τέλος, δεν θα μπορούσα να μην εκφράσω την ευγνωμοσύνη μου στους γονείς μου, που με αυτοθυσία και αγάπη με υποστηρίζουν όλα αυτά τα χρόνια, καθώς και σε όλους τους κοντινούς μου ανθρώπους για τις όμορφες στιγμές που μου έχουν χαρίσει και εξακολουθούν να μου χαρίζουν.

Χρήστος Κατσακιώρης
Οκτώβριος 2018

Contents

List of figures

List of tables

Εκτενής Περίληψη

Εισαγωγή

Κίνητρα

Σήμερα, στην εποχή της Πληροφορίας, πολλαπλές εκφάνσεις της οικονομικής και κοινωνικής ανάπτυξης άγονται σε μεγάλο βαθμό από τη Βιομηχανία της Πληροφορικής. Μεγάλος όγκος δεδομένων παράγεται και χρησιμοποιείται για την εξυπηρέτηση καθημερινών αναγκών, τόσο ιδιωτικού, όσο και επιχειρηματικού ενδιαφέροντος. Μάλιστα, τα τελευταία χρόνια γινόμαστε μάρτυρες μίας ιστορικής ενίσχυσης αυτού του φαινομένου, το οποίο από τεχνολογική άποψη εγείρει πρωτόγνωρες δυσκολίες αναφορικά με τον όγκο των δεδομένων και την ανάγκη για την αποδοτική τους διαχείριση.

Τέτοιου είδους δυσκολίες εκτείνονται στις περισσότερες κλασικές συνιστώσες των υπολογιστικών συστημάτων: την επεξεργασία, τη δικτύωση και την αποθήκευση. Συνεπακολούθως, έχουν επιφέρει την ανάπτυξη επιλύσεων που εκμεταλλεύονται τη Θεωρία των Κατανεμημένων Συστημάτων μόνο σχετικά πρόσφατα, παρά την διαρκή παρουσία και εξέλιξη της τελευταίας επί δεκαετίες, ομολογουμένως σε θεωρητικό επίπεδο κατά κύριο λόγο. Με αυτόν τον τρόπο, τα υποδείγματα επεξεργασίας και διαχείρισης δεδομένων στρέφονται σε κατανεμημένες εφαρμογές και συστήματα, διαμορφώνοντας ένα νέο πεδίο έρευνας και πιθανών βελτιστοποιήσεων, συχνά συνυφασμένο με τον όρο “Big Data”[1].

[1] Η πιστή μετάφραση του όρου “Big Data” είναι “Μεγάλα Δεδομένα”, αλλά ελεύθερα θα αποδίδονταν καλύτερα ως “Δεδομένα Μεγάλου Όγκου”. Εν προκειμένω, διατηρήθηκε ο ευρέως διαδεδομένος αγγλικός όρος για την έκφραση της ίδιας έννοιας.

Η απαιτούμενη υλική υποδομή που καλείται να φιλοξενήσει τα κατανεμημένα αυτά συστήματα, αλλά και τον όγκο της πληροφορίας που εισρέει και εκρέει προς και από αυτά, αποτελείται συνήθως από κέντρα δεδομένων, συχνά περισσότερα του ενός, συνδεδεμένα μεταξύ τους και πιθανώς διασκορπισμένα σε πολλά μέρη του κόσμου. Επιπροσθέτως, προκειμένου να μειωθεί το κόστος της εγκατάστασής τους, αλλά και προς αποφυγήν κινδύνων σχετικών με δεσμεύσεις στη χρήση προϊόντων συγκεκριμένων εταιρειών, το ρεύμα της "ευρείας εμπορευματοποίησης" των υποδομών υλικού[2] στα σύγχρονα κέντρα δεδομένων συνεχώς κερδίζει έδαφος.

Εντωμεταξύ, η ταχύρρυθμη επικράτηση των προτύπων Μικροϋπηρεσιών στη σύγχρονη αρχιτεκτονική λογισμικού, έχει ακουσίως οδηγήσει σε μία αδιάλειπτη διαδικασία επανεξετάσεων των διαθέσιμων μοντέλων για deployment[3] εφαρμογών. Η χρήση παραδοσιακών εικονικών μηχανών, πάλαι ποτέ η συνήθης επιλογή ως στοιχείο deployment στα κέντρα δεδομένων, σταδιακά αντικαθίσταται, ή συνδυάζεται, με περιέκτες[4]. Αυτή η νέα τεχνολογία, βασισμένη σε έννοιες και κατασκευάσματα που υπάρχουν εδώ και αρκετά χρόνια, αλλά έγιναν δημοφιλή για πρώτη φορά από το Docker, περίπου το 2013, συνιστά σήμερα μία ευρέως θεωρούμενη ως βέλτιστη επιλογή για τον ανωτέρω σκοπό. Ως μία από τις πιο πρόσφατες και "χειμαρρώδεις" τάσεις της βιομηχανίας, ρηξικέλευθα συστήματα αναπτύσσονται ενεργά και συνεχώς για την υποστήριξή της, διαμορφώνοντας έτσι γύρω της ένα νέο, πλούσιο οικοσύστημα με πολλά υποσχόμενα χαρακτηριστικά.

Καθώς μέρα με τη μέρα οι περιπτώσεις εφαρμογής της περιεκτικοποίησης[5] πληθαίνουν, η κοινή μεταξύ όλων ανάγκη για αντίστοιχα αποδοτικά εργαλεία διαχείρισης καθίσταται εντονότερη, και συνεπώς καταλαμβάνει ένα μεγάλο, και διαρκώς αυξανόμενο, μερίδιο του προαναφερθέντος οικοσυστήματος. Συστήματα ανοικτού κώδικα, όπως τα fleet και Container Linux της CoreOS, Inc. (πλέον της Red Hat), και το Kubernetes, "γεννημένο" στη Google, είναι μερικά από τα πρώτα συστήματα για διαχείριση και ενορχήστρωση deployment με που έχουν ως δομικό τους στοιχείο πε-

[2]Με τη φράση "ευρεία εμπορευματοποίηση υποδομών υλικού" στο εξής θα αναφερόμαστε στην έννοια που εκφράζεται με την αγγλική φράση "commoditization of the hardware infrastructure".

[3]Ελλείψει κατάλληλου όρου, γίνεται χρήση του αγγλικού "deployment" αντί του εν προκειμένω αδόκιμου ελληνικου όρου "παράταξη".

[4]Ελλείψει αντίστοιχου όρου στην ελληνική βιβλιογραφία, χρησιμοποιείται ο ελληνικός όρος "περιέκτης" για να εκφράσει την έννοια που εκφράζεται με τον αγγλικό όρος "container".

[5]Απουσία αντίστοιχου όρου στην ελληνική βιβλιογραφία, χρησιμοποιείται ο ελληνικός όρος "περιεκτικοποίηση" για να εκφράσει την έννοια που εκφράζεται με τον αγγλικό όρο "containerization".

ριέκτες. Από τη διάδοσή τους, και την επερχόμενη αναγνώριση της αξίας της λειτουργίας τους, η ποικιλία των διαθέσιμων σχετικών επιλογών έχει αυξηθεί, καθώς αρκετά περισσότερα τέτοια συστήματα, είτε ανοικτού είτε κλειστού κώδικα, έχουν είτε κτιστεί εξαρχής ή χρησιμοποιώντας άλλα υπάρχοντα ως βάση, είτε υπάρχοντα συστήματα έχουν επαναπροσαρμοστεί με τρόπο ώστε να περιλαμβάνουν την υποστήριξη των περιεκτών. Παραδείγματα αυτών των κατηγοριών αποτελούν τα Tectonic της CoreOS, Docker Swarm της Docker, OpenShift της Red Hat, Apache Mesos, DC/OS της Mesosphere, αλλά και πολλά ακόμη, τα οποία, όμως, δεν έχουμε στόχο να τα απαριθμήσουμε στο σύνολό τους.

Στην παρούσα διπλωματική, εστιάζουμε αποκλειστικά στο Kubernetes, χρησιμοποιώντας το Docker στο στρώμα περιεκτικοποίησής του. Αυτή την περίοδο λαμβάνουν χώρα σημαντικές προσπάθειες για την ανάπτυξη του Kubernetes, και την ανάδειξή του ως την θεμελιώδη και ευρέως χρησιμοποιούμενη πλατφόρμα για τη διαχείριση περιεκτικοποιημένων[6] deployment εφαρμογών σε περιβάλλοντα Cloud, είτε αυτά αποτελούν ιδιοκτησία της οντότητας που τα χρησιμοποιεί, είτε μισθώνονται από τρίτες οντότητες. Η κοινότητα του Kubernetes διευρύνεται με ταχύτατους ρυθμούς, αφού όλο και περισσότεροι μηχανικοί και προγραμματιστές συμμετέχουν σ' αυτή, είτε αυτόβουλα, είτε μέσω εταιρειών στις οποίες αυτοί εργάζονται – πολλές τέτοιες εταιρείες, δε, είναι κολοσσοί της παγκόσμιας βιομηχανίας της Πληροφορικής, όπως οι Google, Red Hat, Microsoft, IBM, Intel, VMWare, Alibaba, Huawei, HP, Cisco Systems, Samsung SDS, Oracle, AT&T, και άλλες. Γνώμη των γραφόντων είναι ότι όλο αυτό δεν συνιστά απλώς μία παροδική τάση, αλλά ότι, απεναντίας, το Kubernetes πρόκειται σύντομα να αποτελέσει το de facto καθιερωμένο εργαλείο για τον σκοπό που εξυπηρετεί.

Η έννοια της απουσίας μόνιμης κατάστασης μίας υπηρεσίας πηγάζει από τις σχεδιαστικές αρχές της Υπηρεσιοστρεφούς Αρχιτεκτονικής[7]. Σύμφωνα με το [10], σε ελεύθερη απόδοση, "μία υπηρεσία μπορεί να είναι γενικώς ενεργή, αλλά όχι ενεργώς εμπλεκόμενη στην επεξεργασία ή διαχείριση δεδομένων μόνιμης κατάστασης. Τέτοιες υπηρεσίες λέγονται stateless. Εντελώς αντίστοιχα, υπηρεσίες που ενεργώς επεξεργάζονται ή διατηρούν δεδομένα μόνιμης κατάστασης λέγονται stateful". Η ίδια συλλογιστική

[6]Ελλείψει αντίστοιχου ελληνικού όρου, χρησιμοποιείται ο ενδεχομένως αδόκιμος ελληνικός όρος "περιεκτικοποιημένος" για να εκφράσει την έννοια του αγγλικού όρου "containerized".

[7]Με τον ελληνικό όρο "Υπερεσιοστρεφής Αρχιτεκτονική" περιγράφεται το σύνολο των σχεδιαστικών τεχνικών και μεθόδων που περιγράφει ο αγγλικός όρος "Service Oriented Architecture" ή "SOA".

ισχύει και στην περίπτωση της Αρχιτεκτονικής Μικροϋπηρεσιών:

- "**stateless**" μικροϋπηρεσίες είναι εκείνες που ούτε επεξεργάζονται ούτε αποθηκεύουν δεδομένα μόνιμης κατάστασης, παρότι ενδεχομένως επεξεργάζονται ή και αποθηκεύουν προσωρινά δεδομένα. Χαρακτηριστικά παραδείγματα τέτοιων αποτελούν οι διεπαφές χρηστών[8], οι οποίες πολύ συχνά βασίζονται στο πρωτόκολλο HTTP[9], το οποίο μάλιστα από τη σχεδίασή του δεν διατηρεί κατάσταση (είναι "stateless").

- "**stateful**" μικροϋπηρεσίες και εφαρμογές είναι εκείνες για τις οποίες η διατήρηση κάποιου είδους μόνιμης κατάστασης είναι ζωτικής σημασίας για το είδος της λειτουργίας και των δυνατοτήτων που παρέχουν. Παραδείγματα εφαρμογών αυτής της κατηγορίας είναι όλες οι βάσεις δεδομένων καθώς και κάθε είδους εφαρμογή ή υποσύστημα που βασίζεται σε δοσοληψίες[10].

Μία εναλλακτική θεώρηση για τη διάκριση των δύο αυτών κατηγοριών υπηρεσιών παρουσιάζεται στο [11], σύμφωνα με το οποίο (σε ελεύθερη απόδοση), οι "stateless" υπηρεσίες ή εφαρμογές είναι συχνά "υπηρεσίες που συγκεντρώνουν απαντήσεις από άλλες υπηρεσίες", ενώ οι "stateful" υπηρεσίες ή εφαρμογές είναι εκείνες που συνήθως "παράγουν απαντήσεις μέσω της εκτέλεσης αλγορίθμων εξειδικευμένης λογικής με δεδομένα μόνιμης κατάστασης".

Ένα από τα πλέον χρήσιμα ποιοτικά χαρακτηριστικά των εφαρμογών που δεν διατηρούν μόνιμη κατάσταση, είναι η φυσικότητα και η μεγάλη ευκολία στην κλιμακωσιμότητά τους. Η "οριζόντια κλιμακωσιμότητά" τους συνήθως μπορεί να επέλθει απλά θέτοντας περισσότερα εκτελέσιμα προγράμματα της εφαρμογής σε λειτουργία, με τη μορφή διεργασιών, ενδεχομένως διασκορπισμένων σε πολλαπλούς κόμβους μιας συστοιχίας, πολύ πιθανόν κάνοντας χρήση ενός κυκλικής λογικής ισοκατανεμητή φόρτου[11], ενώ το σύνολο των αναγκών τους για αποθήκευση δεδομένων καλύπτεται με

[8] Το ελληνικό ονοματικό σύνολο "διεπαφή χρηστών" χρησιμοποιείται εν προκειμένω για την απόδοση του αγγλικού "User Interface" ή "UI".

[9] Το HTTP είναι ακρωνύμιο για το πρωτόκολλο "Hypertext Transfer Protocol" ή "Πρωτόκολλο Μεταφοράς Υπερκειμένου".

[10] Ο ελληνικός όρος "δοσοληψία" χρησιμοποιείται για να εκφράσει την έννοια που κατέχει στη διεθνή βιβλιογραφία του πεδίου των Βάσεων Δεδομένων ο αγγλικός όρος "transaction".

[11] Το ονοματικό σύνολο "κυκλικής λογικής ισοκατανεμητής φόρτου" αποτελεί μία προσπάθεια έκφρασης της έννοιας του αντίστοιχου αγγλικού "round-robin load balancer".

χρήση χαμηλής χωρητικότητας, ευφήμερου τύπου αποθηκευτικό χώρο. Αυτή η λογική είναι ευθυγραμμισμένη με εκείνη της κοινής αρχιτεκτονικής των περισσότερων μηχανών περιεκτών (λόγου χάριν [12]). Ένας περιέκτης αποτελείται από πολλαπλά αμετάβλητα στρώματα ανάγνωσης-μόνο, τα οποία συνδιαμορφώνουν μία "εικόνα", και οποιοδήποτε υποσύνολο αυτών μπορεί να αποτελεί αντικείμενο διαμοιρασμού από πολλούς περιέκτες ταυτόχρονα. Ένα απλό, "ελαφρύ", εγγράψιμο στρώμα εναποτίθεται στην κορυφή των παραπάνω, και όπου αποθηκεύεται οποιαδήποτε προσωρινή τροποποίηση δεδομένων. Κάθε περιέκτης έχει το δικό του τέτοιο εγγράψιμο στρώμα, το οποίο καταστρέφεται μόνο ταυτόχρονα με την καταστροφή του ίδιου του περιέκτη. Καθίσταται πασιφανές ότι οι εφαρμογές οι οποίες δεν έχουν την ανάγκη διατήρησης μόνιμης κατάστασης, εν τέλει συνιστούν τους ιδανικούς υποψηφίους για περιεκτικοποιημένα "deployment" εφαρμογών. Πράγματι, τις περισσότερες φορές αυτές αποτελούν το σημείο εκκίνησης των πειραματισμών περί περιεκτικοποίησης σε περιβάλλοντα παραγωγής, για πληθώρα οργανισμών.

Παρ᾽ολ᾽αυτά, ένα καίριο σχετικό πρόβλημα το οποίο τίθεται, μεταξύ άλλων, στο [13], είναι ότι "σπάνια υπάρχει αρχιτεκτονική χωρίς ανάγκη για διατήρηση μόνιμης κατάστασης". Στη συντριπτική τους πλειοψηφία, οι υπηρεσίες πράγματι χρειάζονται πληροφορία μόνιμης κατάστασης *κάπου*, προκειμένου να επιτελέσουν οποιαδήποτε χρήσιμη ενέργεια, συνεπώς πρέπει να υπάρχει τουλάχιστον ένα στοιχείο της αρχιτεκτονικής που να καλύπτει την ανάγκη μόνιμης αποθήκευσης. Πώς όμως υποτίθεται ότι θα έπρεπε να αντιμετωπίζεται αυτό το ζήτημα στο σύγχρονο κόσμο των περιεκτικοποιημένων μικροϋπηρεσιών; Η Ετήσια Έρευνα Υιοθέτησης Περιεκτών της Portworx[12][14], αποκαλύπτει, μεταξύ άλλων ενδιαφέροντων στοιχείων και γεγονότων, ότι η μόνιμη αποθήκευση για περιέκτες παραμένει ένα από τα πλέον δυσεπίλυτα προβλήματα ακόμα και σήμερα, και ότι δεν έχει ακόμα ουσιαστικά επιλυθεί. Μάλιστα, ακόμα και το The Twelve-Factor App[15], μία ευρέως διαδεδομένη μεθοδολογία για ανάπτυξη σύγχρονων εφαρμογών Cloud, κατηγοριοποιεί "βολικά" όλες τις εφαρμογές με ανάγκη για διατήρηση μόνιμης κατάστασης, συμπεριλαμβανομένων των συστημάτων βάσεων δεδομένων, μηνυμάτων, αναμονής και προσωρινής αποθήκευσης, υπό τον όρο "Υποβοηθητικές Υπηρεσίες", προτείνοντας τον χειρισμό τους ως επιπρόσθετους εξωγενείς πόρους, ουσιαστικά διαχωρίζοντάς τα από μία σχεδίαση αμιγώς βασιμένη στο Cloud.

Είναι αξιοσημείωτο δε, ότι οι εφαρμογές που έχουν την ανάγκη διατήρησης μόνι-

[12]Portworx Annual Container Adoption Survey 2017 [14].

μης κατάστασης είναι εγγενώς αρκετά δυσκολότερο να κλιμακωθούν οριζοντίως σε σύγκριση με αυτές που δεν έχουν τέτοια ανάγκη, και ιδίως όσες σχεδιάστηκαν και υλοποιήθηκαν χωρίς τον συνυπολογισμό της ύπαρξης περιεκτών, όπως, λόγου χάριν, παραδοσιακά συστήματα βάσεων δεδομένων, και οι οποίες απαιτούν ιδιαίτερη μεταχείριση, συχνά εξατομικευμένη ανά περίπτωση. Κάθε στιγμιότυπο τέτοιων εφαρμογών (είτε πρόκειται για διεργασία είτε για κάποιο σύνολο διεργασιών του συστήματος), συνήθως έχει μία μοναδική ταυτότητα, και αυστηρώς καθορισμένο ρόλο στο σύστημα, με την έννοια ότι πολλαπλά τέτοια στιγμιότυπα δεν είναι εναλλάξιμα. Αυτή η ταυτότητα μπορεί να είναι, επί παραδείγματι, συνυφασμένη σε μία συγκεκριμένη οντότητα αποθήκευσης (όπως κάποιον τόμο), και στην οποία η εφαρμογή πολύ πιθανόν να χρειάζεται να επιτελέσει κάποιου είδους διαδικασία ανάκτησης κατά την επαναφορά του από σφάλματα, ή μπορεί να είναι συνδεδεμένη με κάποιον συγκεκριμένο ρόλο σε ένα κατανεμημένο πρωτόκολλο (λόγου χάριν, ένα εξ αυτών που παρέχουν κατανεμημένη συμφωνία).

Πριν την έκρηξη της περιεκτικοποίησης, καθοδηγούμενο από την ευρεία εμπορευματοποίηση των υποδομών υλικού, αλλά ταυτόχρονα ενισχύοντάς την, ένα άλλο κίνημα, αυτό της προγραμματιζόμενης αποθήκευσης[13], ήδη σταδιακά επικρατεί.

Οι παραδοσιακές λύσεις για την αποθήκευση στα κέντρα δεδομένων είναι τα SANs. Η χρήση ιδιωτικού, κλειστού κώδικα λογισμικού που εκτελείται σε ιδιωτικές, κλειστού τύπου υποδομές υλικού συναποτελούμενες από πολλαπλές συστοιχίες δίσκων και ελεγκτών, ήταν μέχρι πρόσφατα ο κανόνας, και εξακολουθεί να επικρατεί σε μεγάλο μερίδιο της αγοράς. Παρ᾽ολ᾽αυτά, οι οργανισμοί που χρησιμοποιούν τέτοιου είδους συστήματα συχνά δυσκολεύονται να προσαρμοστούν στην εκθετική αύξηση των δεδομένων τους, παρότι και αυτά τα συστήματα διαρκώς εξελίσσονται επιδιώκοντας να προσαρμοστούν στις σύγχρονες συνθήκες των Δεδομένων Μεγάλου Όγκου.

Ως εναλλακτική, πληθώρα καινοτόμων κατανεμημένων συστημάτων αποθήκευσης έχουν αναπτυχθεί κατά την τελευταία δεκαετία, και μερικά εξ αυτών έχουν ήδη αποδείξει την αξία τους ως προς την αξιοπιστία και τη στιβαρότητά τους σε περιβάλλοντα παραγωγής. Αντί να στηρίζονται σε ταχύτερους δίσκους ή σε υψηλότερο εύρος ζώνης των δικτύων, επιστρατεύουν διαχειριστικές τεχνικές βασιζόμενες σε Διεπα-

[13] Με τον ελληνικό όρο "προγραμματιζόμενη αποθήκευση" αναφερόμαστε στον αγγλικό "software-defined storage".

φές Προγραμματισμού Εφαρμογών[14] ώστε να επιτελέσουν παραδοσιακές λειτουργίες της αποθήκευσης, όπως προετοιμασία, διαχείριση και παρακολούθηση, με τρόπο προγραμματιστικό, σε απλούς ευρέως εμπορευματοποιημένους δίσκους. Με τους απλούς δίσκους του εμπορίου να γίνονται συνεχώς φθηνότεροι και συνεπώς εύκολότερα αντικαταστάσιμοι, και πιο αποδοτικοί (λόγου χάριν, εξαιτίας της καινούργιας τεχνολογίας NVM[15], όπως SSDs[16], και προσφάτως σε συνδυασμό με NVMe[17], αλλά κυρίως ανεξάρτητοι από συγκεκριμένες πλατφόρμες ή κλειστούς εξοπλισμούς και υλικό αποθήκευσης, τέτοια συστήματα επιτυγχάνουν συνέπεια και πλεονασμό των δεδομένων, καθώς και υψηλή διαθεσιμότητα ακόμα και σε περιπτώσεις διαμερίσεων δικτύου, χρησιμοποιώντας ευφυείς αλγορίθμους και έννοιες της Θεωρίας των Κατανεμημένων Συστημάτων.

Αντί να προσπαθήσουμε να απαριθμήσουμε τα σημαντικότερα εξ αυτών των κατανεμημένων συστημάτων αποθήκευσης, το οποίο ούτως ή άλλως θα ήταν αμφιλεγόμενο, αφού πολλά από αυτά εισήγαγαν χαρακτηριστικά που αργά ή γρήγορα επηρέασαν και τα υπόλοιπα, επιλέγουμε να αναφέρουμε μόνο δύο από αυτά, τα οποία εν μέρει αποτέλεσαν και δύο από τις πηγές έμπνευσης για τις σχεδιάσεις των συστημάτων αυτής της διπλωματικής: το Ceph[16, 17, 18] και το Cassandra[19, 20].

Διατύπωση Προβλήματος

Στην παρούσα εργασία εστιάζουμε στη διαλειτουργικότητα των περιεκτών και της τοπικής μόνιμης αποθήκευσης, στον σύγχρονο κόσμο των μικροϋπηρεσιών σε περιβάλλοντα Cloud.

Ισχυριζόμαστε ότι τα κατανεμημένα συστήματα αποθήκευσης πρέπει να εκμεταλλεύονται στο έπακρο την πληθώρα πλεονεκτημάτων που παρέχει η χρήση περιεκτών για την ανάπτυξη, διανομή και εκτέλεση εφαρμογών σε περιβάλλοντα υποδομών Cloud, και την χρηστικότητα των χαρακτηριστικών που προσφέρουν τα ποικίλα εργαλεία ενορχήστρωσής τους, και γενικότερα το ακμάζον οικοσύστημά τους.

[14]Με τον ελληνικό όρο "Διεπαφή Προγραμματισμού Εφαρμογών" αναφερόμαστε στον αντίστοιχο αγγλικό "Application Programming Interface" ή "API".

[15]Το ακρωνύμιο "NVM" αναφέρεται στον αγγλικό όρο "Non-volatile memory", ο οποίος αποδίδεται με τον ελληνικό "Μη-πτητική μνήμη".

[16]Το ακρωνύμιο "SSD" αναφέρεται στον αγγλικό όρο "Solid State Drive", ο οποίος αποδίδεται με τον ελληνικό "Δίσκος Στεράς Κατάστασης".

[17]Το ακρωνύμιο "NVMe" αναφέρεται στον όρο "NVM Express".

Πρωταρχικός στόχος μας είναι να εξετάσουμε τον τρόπο με τον οποίο μπορούμε να εξασφαλίσουμε χαρακτηριστικά μόνιμης αποθήκευσης μέσω τοπικών δίσκων, φυσικά, χρησιμοποιώντας ως πλατφόρμα και εργαλεία τα Kubernetes και Docker. Εξερευνούμε τον τρόπο με τον οποίον η Διεπαφή Προγραμματισμού Εφαρμογών του Kubernetes μπορεί να εξυπηρετήσει ένα σύνολο "ευφυών" περιεκτικοποιημένων agents και proxy servers [18], επιτρέποντάς τους εν τέλει να υποστηρίξουν μερικά εκ των ων ουκ άνευ χαρακτηριστικών των σύγχρονων κατανεμημένων συστημάτων αποθήκευσης, όπως κλιμακωσιμότητα και ελαστικότητα, ισοκατανομή φόρτου, αυτοΐαση σε περιπτώσεις σφαλμάτων, καθώς και υψηλή διαθεσιμότητα και ανοχή σε σφάλματα μέσω αυτοματοποιημένης δημιουργίας αντιγράφων των δεδομένων.

Γι᾽αυτόν τον λόγο, σχεδιάζουμε και υλοποιούμε ένα κατανεμημένο σύστημα αποθήκευσης αντικειμένων, απλής αλλά ορθής λειτουργίας, σύμφωνα με τις αρχές που ορίζουν τα πρότυπα Αρχιτεκτονικής Μικροϋπηρεσιών. Για να αντιμετωπίσουμε την μεγάλη πολυπλοκότητα που εισάγουν οι πράξεις μεταβολής των δεδομένων (update operations) ως προς την συνέπεια των δεδομένων που αποθηκεύονται στα κατανεμημένα συστήματα αποθήκευσης γενικώς, χωρίς όμως να παρεκκλίνουμε και από τον κύριο στόχο αυτής της εργασίας, εκμεταλλευόμαστε έννοιες όπως η αμεταβλητότητα των δεδομένων σε συνδυασμό με την διευθυνσιοδότηση βάσει περιεχομένου, ο κατακερματισμός, και η κατάτμηση του χώρου κλειδιών του συστήματος, έννοιες για τις οποίες πρόκειται να συζητήσουμε παρακάτω.

Η πληροφορία αποθηκεύεται στη μορφή μεγάλων δυαδικών αντικειμένων[19] αναγνωρίσιμων με μοναδικά κλειδιά. Το κλειδί του αντικειμένου είναι και ο μοναδικός τρόπος προσπέλασής του: υιοθετώντας τις βασισμένες στην απλότητα αρχές σχεδίασης που προωθούνται από το μοντέλο NoSQL[20], και σε αντίθεση με τα παραδοσιακά Σχεσιακά Συστήματα Διαχείρισης Βάσεων Δεδομένων[21], περίπλοκες αναζητήσεις, ευρετηριοποίηση και σύνθετες λειτουργίες διαχείρισης θεωρούνται περιττές για τον σκοπό μας, και γι᾽αυτό δεν υποστηρίζονται.

[18]Οι αγγλικοί όροι "agent" και "proxy server" προτιμώνται – μόνο προς το παρόν – έναντι των αντίστοιχων ελληνικών "πράκτορας" ή "ενεργών" και "μεσολαβητικός διακομιστής", προς αποφυγήν εμφιλοχώρησης αποπροσανατολιστικών εννοιών στον ειρμό του αναγνώστη.

[19]Με τον ελληνικό όρο "μεγάλο δυαδικό αντικείμενο" αναφερόμαστε στον αντίστοιχο αγγλικό "binary large object" ή "blob".

[20]Πρόκειται για το μοντέλο "Not-only SQL".

[21]Με το ελληνικό ονοματικό σύνολο "Σχεσιακό Σύστημα Διαχείρισης Βάσης Δεδομένων" εκφράζουμε το αντίστοιχο αγγλικό "Relational Database Management System" ή "RDBMS".

Η αρχική μας σχεδίαση βασίζεται σε ένα στατικό σχήμα κατάτμησης του πεδίου κλειδιών. Εμπνευσμένο από το RADOS[22][17] του Ceph, το πεδίο κλειδιών κατατέμνεται σε ένα στατικό πλήθος τμημάτων, τα οποία αντιστοιχίζονται σε σύνολα περιεκτών. Η επιλογή του κατάλληλου τέτοιου τμήματος για κάθε περίπτωση εισερχόμενου αιτήματος ανάγνωσης ή εγγραφής αντικειμένου στο "εμπρόσθιο επίπεδο"[23] του συστήματος, υπολογίζεται εφαρμόζοντας μία πολύ απλή συνάρτηση κατακερματισμού στο κλειδί του εκάστοτε αντικειμένου. Κατόπιν, το αίτημα δρομολογείται προς το κατάλληλο σύνολο από agents στο "οπίσθιο επίπεδο"[24] του συστήματος. Η διαδικασία αυτή είναι παρόμοια με την κατανομή των αντικειμένων μεταξύ συνόλων από Placement Groups[25] που διεξάγεται από το RADOS χρησιμοποιώντας τον αλγόριθμο CRUSH[26][18].

Εξαιτίας ζητημάτων κλιμακωσιμότητας που προέκυψαν από την παραπάνω σχεδίαση, εν μέρει οφειλόμενα και στο Kubernetes, κατά τον δεύτερο κύκλο σχεδίασης χρησιμοποιείται ο αλγόριθμος "συνεπούς κατακερματισμού"[27][21] μεταξύ των agents στο "οπίσθιο επίπεδο" του συστήματος, διατηρώντας ταυτόχρονα την υπόλοιπη αρχιτεκτονική του συστήματος όσο το δυνατόν περισσότερο αμετάλλακτη. Ο αλγόριθμος συνεπούς κατακερματισμού έχει τεθεί σε εφαρμογή στο επανειλημμένα κατά το παρελθόν, από ένα πλήθος κατανεμημένων συστημάτων αποθήκευσης, όπως τα: Dynamo[11] της Amazon, Cassandra[19, 20], την υπηρεσία αποθήκευσης αντικειμένων Swift[22] του Openstack, το Riak KV[23] της Basho, τα Voldemort[24] και GlusterFS[25], και άλλα, και συνεπώς θεωρείται ως μία αξιόπιστη και δοκιμασμένη τεχνική αντιμετώπισης τέτοιου είδους δυσκολιών.

Το τελικό αποτέλεσμα είναι μία γραμμικώς κλιμακώσιμη, αυτοϊούμενη αρχιτεκτονική υψηλής διαθεσιμότητας για ένα κατανεμημένο σύστημα αποθήκευσης αντικειμένων με αντιγραφή, για Cloud περιβάλλοντα. Όντας, προς το παρόν, υλοποιημένο ως επέκταση της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes, απλώς προσαρ-

[22]Reliable, Autonomic Distributed Object Store.

[23]Ο ελληνικός όρος "εμπρόσθιο επίπεδο", αναφερόμενος στο σύστημα, εκφράζει τον αγγλικό "frontend tier".

[24]Ο ελληνικός όρος "οπίσθιο επίπεδο", αναφερόμενος στο σύστημα, εκφράζει τον αγγλικό "backend tier".

[25]Ο αγγλικός όρος "Placement Group", τεχνική ορολογία του συστήματος Ceph, θα μπορούσε να αποδοθεί με τον ελληνικό "Ομάδα Τοποθέτησης".

[26]Ακρωνύμιο για "Controlled Replication Under Scalable Hashing".

[27]Ο ελληνικός όρος "συνεπής κατακερματισμός" αναφέρεται στον αλγόριθμο που περιγράφεται με τον αγγλικό όρο "consistent hashing".

τώντας του μία κατάλληλη μηχανή αποθήκευσης[28] θα επέτρεπε στο τελικό μας σύστημα να λειτουργήσει ως ένα αξιοπρεπές κατανεμημένο σύστημα αποθήκευσης αντικειμένων για περιπτώσεις απλών απαιτήσεων. Διαφορετικά, μπορεί να θεωρηθεί ως ένα ενδιάμεσο στρώμα ενός μεγαλύτερο κατανεμημένου συστήματος αποθήκευσης, το οποίο θα παρείχε, εκτός της κατάλληλης μηχανής αποθήκευσης ως ένα χαμηλότερο στρώμα, και ένα υψηλότερο στρώμα υπεύθυνο ενδεχομένως για τη διαχείριση διαφορετικών εκδόσεων των αντικειμένων, ώστε να υλοποιούνται με αυτόν τον τρόπο και πράξεις μεταβολής τους.

Υπάρχουσες Λύσεις

Το αντικείμενο της παρούσας εργασίας, η χρήση τοπικών, ευρέως εμπορευματοποιημένων δίσκων για την παροχή μόνιμης αποθήκευσης σε εφαρμογές που διαχειρίζεται το Kubernetes και έχουν ανάγκη αποθήκευσης μόνιμης κατάστασης, δεν αποτελεί ζήτημα που δεν έχει τεθεί ξανά στο παρελθόν. Πολλές προσπάθειες έχουν πραγματοποιηθεί, και πολλά έργα και προϊόντα εξελίσσονται ενεργά με σκοπό την αντιμετώπισή του. Παρ᾽ολ᾽αυτά, κανένα εξ αυτών μέχρι στιγμής δεν έχει καταφέρει να χαράξει έναν αδιαμφισβήτητα προτιμότερο τρόπο αντιμετώπισής του, σε βαθμό που να έχει επικρατήσει στην αγορά.

Οι περισσότερες από τις υπάρχουσες λύσεις είναι ουσιαστικά προσπάθειες προσαρμογής των deployment υπαρχόντων κατανεμημένων συστημάτων αποθήκευσης στους όρους που καθορίζει το Kubernetes. Ομολογουμένως, η αξιοπιστία και η σταθερότητα που παρέχουν συστήματα όπως τα Ceph και Cassandra, τα οποία αναπτύσσονται πολλά χρόνια και έχουν επανειλημμένα δοκιμαστεί σε περιπτώσεις περιβαλλόντων παραγωγής με επιτυχία, είναι δελεαστικά χαρακτηριστικά για ένα τόσο σημαντικό πρόβλημα, όπως είναι η αποθήκευση δεδομένων. Όμως, έχει υπάρξει τουλάχιστον μία σοβαρή προσπάθεια κατασκευής, από το μηδέν, ενός κατανεμημένου συστήματος αποθήκευσης που να χρησιμοποιεί το Kubernetes και τη λογική του ως θεμέλιο λίθο για τη δόμηση και την αρχιτεκτονική του. Το όραμα αυτής της προσπάθειας είναι που διέπει και την παρούσα εργασία.

Παρακάτω, θα περιγράψουμε εν συντομία δύο τέτοιες αξιοσημείωτες υπάρχουσες λύ-

[28] Με τον ελληνικό όρο “μηχανή αποθήκευσης” αναφερόμαστε στην έννοια που εκφράζει η αγγλική ορολογία “storage engine”.

σεις:

- **Torus:** Πρόκειται για ένα ανοικτού κώδικα κατανεμημένο σύστημα αποθήκευσης, το οποίο σχεδιάστηκε και αναπτύχθηκε από την CoreOS με σκοπό να παρέχει αξιόπιστες και κλιμακώσιμες υπηρεσίες αποθήκευσης σε συστοιχίες από περιέκτες, διαχειριζόμενες από το Kubernetes. Όπως το περιγράφει ένας από τους εμπνευστές του, στον πυρήνα του, το Torus είναι μια βιβλιοθήκη με διεπαφή που εμφανίζεται ως παραδοσιακό αρχείο, επιτρέποντας έτσι τη διαχείριση της αποθήκευσης μέσω βασικών και εύληπτων λειτουργιών πάνω σε αρχεία [27]. Υποστηρίζει, ακόμη, την έκθεση αυτού του αρχείου ως αποθήκευση βάσει μπλοκ[29] μέσω NBD[30].

  Το Torus χρησιμοποιεί τον αλγόριθμο συνεπούς κατακερματισμού για την υποστήριξη δημιουργίας αντιγράφων των δεδομένων, περισυλλογής σκουπιδιών και εξισορρόπησης φόρτου, μέσω μιας εσωτερικής χρήσης Διεπαφής Προγραμματισμού Εφαρμογών βασισμένης σε δίκτυο ομότιμων κόμβων. Η σχεδίασή του, επιτρέπει επίσης την υποστήριξη κρυπτογράφησης, αλλά και διόρθωσης σφαλμάτων Reed-Solomon[31], προκειμένου να παράσχει καλύτερη διασφάλιση για την ορθότητα και εμπιστευτικότητα των δεδομένων σε όλη την έκταση του συστήματος.

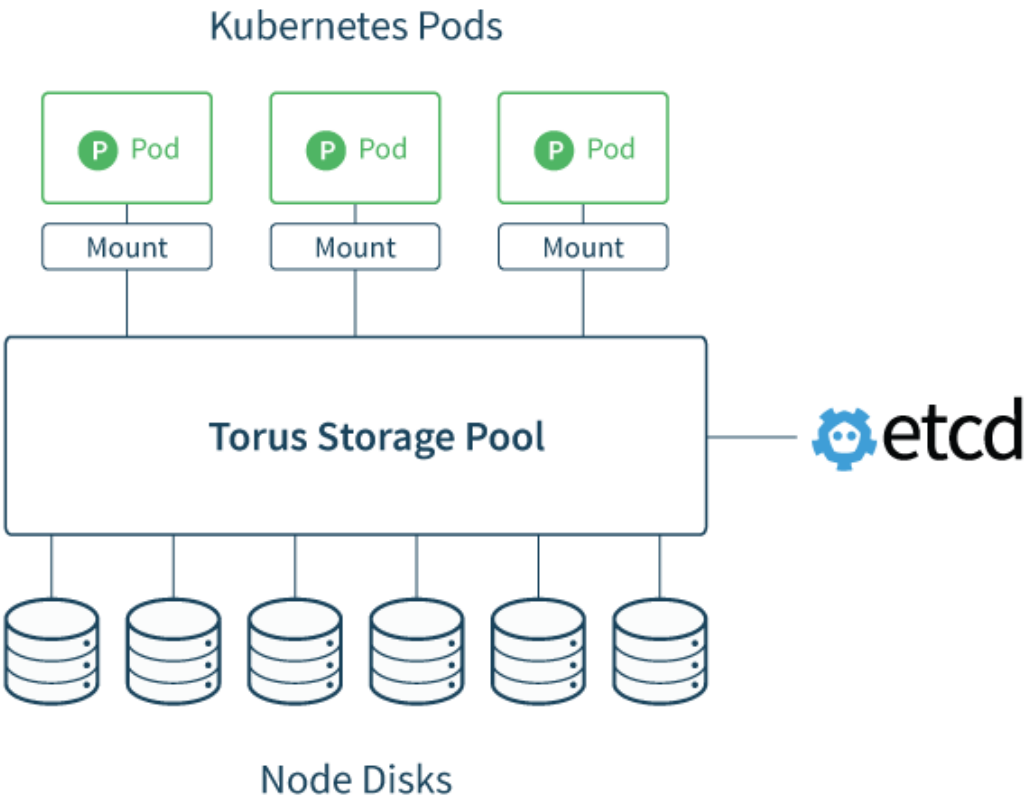
  Δεδομένα τα οποία αφορούν την "κατάσταση του κόσμου", τα οποία συνήθως απαιτούν συμφωνία μεταξύ των ομότιμων κόμβων, αποθηκεύονται στο etcd. Αυτά συμπεριλαμβάνουν πληροφορίες σχετικές με την εξερεύνηση των ομότιμων κόμβων, την συνεργασία τους, στοιχεία για πιθανή μελλοντική ανάκτηση από σφάλματα, μεταδεδομένα χρησιμοποιούμενων τόμων, και άλλα, όμως δεν συμπεριλαμβάνουν τα ίδια τα δεδομένα των χρηστών [29]. Στην αρχιτεκτονική του Torus, το etcd είναι το "σύστημα αποθήκευσης μεταδεδομένων".

  Σύμφωνα με το [29], εκτός του "συστήματος αποθήκευσης μεταδεδομένων", το Torus δομείται χονδροειδώς σε τρία επίπεδα. Το επίπεδο "Διακομιστή" είναι το ανώτατο επίπεδο, με το οποίο αλληλεπιδρούν οι χρήστες του συστήματος για να αναγνώσουν ή να εγγράψουν δεδομένα. Είναι, επίσης, υπεύθυνο για υποσυστήματα συντήρησης της συνδεσιμότητας και παρακολούθησης, και συνεπώς

[29] Με τον ελληνικό όρο "αποθήκευση βάσει μπλοκ" αναφερόμαστε στην έννοια που εκφράζει ο αγγλικός όρος "block-oriented storage".

[30] Ακρωνύμιο για το "Network Block Device".

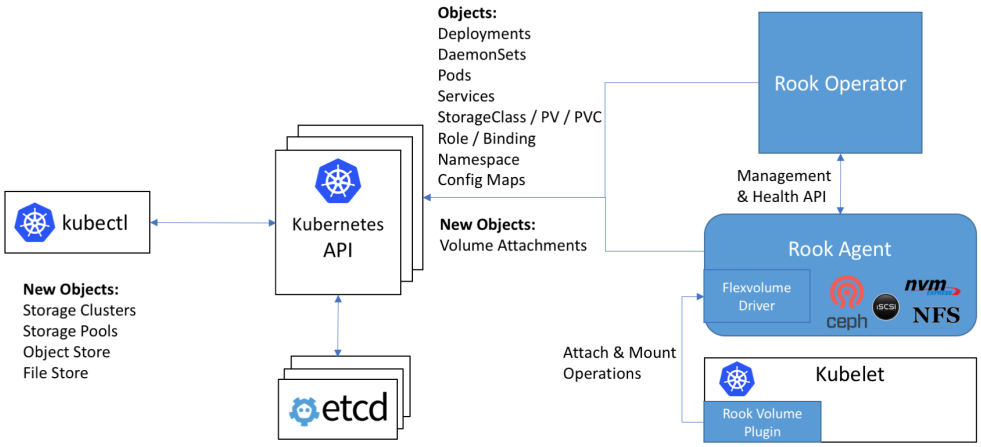
επικοινωνεί με το etcd. Κάτω από αυτό, το επίπεδο "Διανεμητή" διαχειρίζεται την επικοινωνία των ομότιμων κόμβων μέσω gRPC[30], το οποίο είναι ζωτικής σημασίας για την εξισορρόπηση του φόρτου και την περισυλλογή σκουπιδιών. Επιπροσθέτως, παρακολουθεί τα αποθηκευτικά μέσα που έχει στη διάθεσή του στο επόμενο επίπεδο, το επιπεδο "Αποθήκευσης", το οποίο είναι ουσιαστικά ένα σύστημα αποθήκευσης κλειδιού-τιμής, όπου βρίσκονται και τα δεδομένα.



Σχήμα 1: *Το μοντέλο του Torus για την παροχή τοπικής μόνιμης αποθηκευτικής ικανότητας σε Pods του Kubernetes. [26]*

Σύμφωνα με το [26], το Torus ξεκίνησε δοκιμαστικά τον Ιούνιο του 2016 με σκοπό την ανάπτυξη ενός συστήματος αποθήκευσης που είναι εύκολα διαχειρίσιμο από το Kubernetes, και το μοντέλο αυτό αποδείχθηκε υγιές. Παρ'ολ'αυτά, ούτε η ταχύτητα ανάπτυξής του, ούτε το ενδιαφέρον της κοινότητας ήταν αρκετά ικανοποιητικά ώστε να αποτρέψουν την διακοπή της ανάπτυξής του τον Φεβρουάριο του 2017. Παρότι η κοινότητα παροτρύνεται να συνεχίσει το έργο του Torus, δεν υπάρχει καμία ευρέως γνωστή τέτοια προσπάθεια μέχρι στιγμής.

- **Rook:** Πρόκειται για ένα έργο υπό την αιγίδα της CNCF[31] που αποσκοπεί στην αυτοματοποίηση των deployment, εκκίνησης, ρύθμισης και προετοιμασίας, κλιμάκωσης, αναβάθμισης, μετανάστευσης, ανάκτησης από σφάλματα, παρακολούθησης και διαχείρισης πόρων υπαρχόντων κατανεμημένων συστημάτων αποθήκευσης, χρησιμοποιώντας διευκολύνσεις που παρέχονται από το υποκείμενο σύστημα διαχείρισης, ενορχήστρωσης και χρονοδρομολόγησης περιεκτών. Αυτή τη στιγμή είναι σε δοκιμαστικό (alpha) στάδιο, και έχει επικεντρωθεί στην ενορχήστρωση του Ceph πάνω απο το Kubernetes, χρησιμοποιώντας όρους και έννοιες που ορίζονται από το τελευταίο.



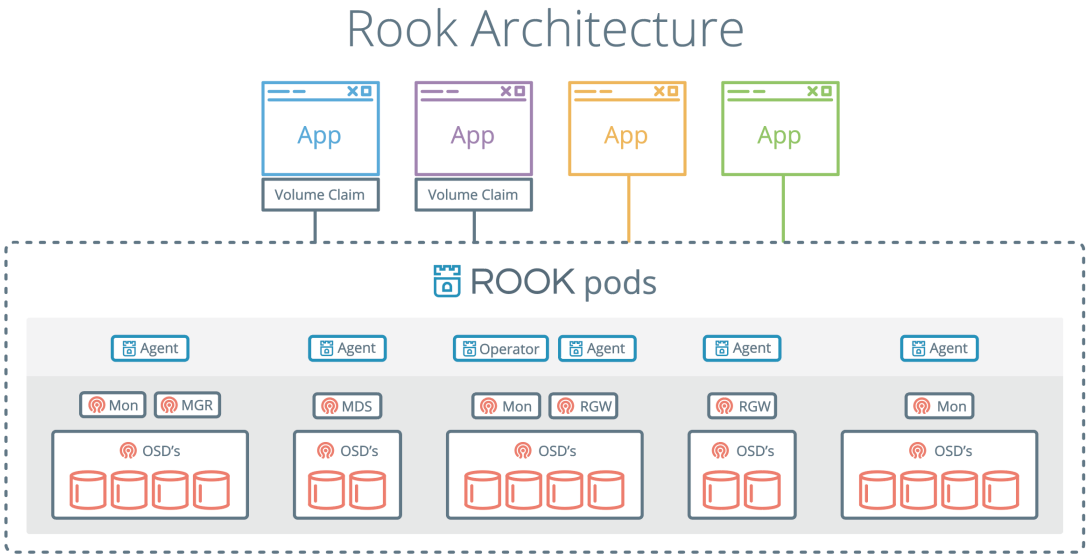
Σχήμα 2: *Η ενσωμάτωση του Rook στο Kubernetes. [32]*

Έχοντας το Rook να εκτελείται σε μία "συστοιχία Kubernetes"[32], εφαρμογές μπορούν να προσπελάσουν συσκευές μπλοκ και συστήματα αρχείων διαχειριζόμενα μέσω του Rook, ή μπορούν να χρησιμοποιήσουν την Διεπαφή Προγραμ-

[31] Ακρωνύμιο για "Cloud Native Computing Foundation", `https://www.cncf.io/`, όπως προβλήθηκε την 24η Απριλίου, 2018.

[32] Στο εξής, με τον ελληνικό όρο "συστοιχία Kubernetes" θα αναφερόμαστε στην έννοια του αγγλικού όρου "Kubernetes cluster", δηλαδή σε μία συστοιχία κόμβων οι οποίοι εκτελούν συντονισμένα το Kubernetes.

ματισμού Εφαρμογών του S3[33]/Swift για αποθήκευση αντικειμένων. Το Rook Operator είναι ένας απλό περιέκτης που εκκινεί και παρακολουθεί Pods του Kubernetes που εκτελούν Ceph Monitors, καθώς και ένα DaemonSet του Kubernetes που διαχειρίζονται την εκτέλεση των Ceph OSDs[35], παρέχοντας βασικές υπηρεσίες αποθήκευσης μέσω του RADOS. Ακόμη, ορίζει, δημιουργεί και διαχειρίζεται CRDs[36] του Kubernetes αντιστοιχίζοντας έννοιες του Ceph σε όρους του Kubernetes, και παρακολουθεί την πορεία τους στο χρόνο κατά το deployment ώστε να διασφαλίσει την συνολική υγεία της συστοιχίας. Τα Ceph Monitors εκκινούνται, ή επανεκκινούνται όποτε κρίνεται αναγκαίο, και γίνεται οποιαδήποτε άλλη προσαρμογή απαιτείται κατά την επέκταση ή συρρίκνωση της συστοιχίας.



Σχήμα 3: *Η αρχιτεκτονική του Rook για το Ceph πάνω από το Kubernetes. [32]*

Η πλήρης διατήρηση της αντιστοιχίας των εννοιών του με το Ceph δεν αποτελεί βασική επιδίωξη για το Rook. Αν και περιλαμβάνει όλους τους αναγκαίους δαίμονες και εργαλεία για την διαχείρηση και αποθήκευση των δεδομένων, πολλές έννοιες του Ceph, όπως τα Placement Groups και CRUSH maps είναι "κρυμμένα" από τους διαχειριστές και τους χρήστες. Αντί αυτού, το Rook πα-

[33]Το Simple Storage Service της Amazon Web Services, ή AWS S3, είναι μία πλατφόρμα Cloud για την αποθήκευση δυαδικών ή αδόμητων δεδομένων [33]. Η Διεπαφή Προγραμματισμού Εφαρμογών του, βασισμένη στις αρχές του REST[34]API, συνήθως χρησιμοποιώντας HTTP, είναι ευρέως δημοφιλής στη βιομηχανία σε ό,τι αφορά συστήματα αποθήκευσης αντικειμένων.

[34]Ακρωνύμιο για τον όρο "Representational State Transfer".

[35]Ακρωνύμιο για το "Object Storage Daemons".

[36]Ακρωνύμιο του "Custom Resource Definitions".

ρέχει μία αρκετά απλουστευμένη εικόνα για τους διαχειριστές της συστοιχίας, οι οποίοι, βέβαια, εξακολουθούν να έχουν πρόσβαση σε πιο προχωρημένες ρυθμίσεις μέσω των εργαλειών του Ceph.

Το Rook είναι ένα έργο σε πολύ μεταγενέστερη και πιο εξελιγμένη κατάσταση από το αποτέλεσμα της παρούσας εργασίας, αφού έχει γίνει συστηματική προσπάθεια για την σχεδίαση και την ανάπτυξή του ώστε να πληροί τις προδιαγραφές των περιβαλλόντων παραγωγής. Απεναντίας, όπως προαναφέρθηκε, κατά την ανάπτυξη του συστήματός μας έχουμε εστιάσει μόνο σε ένα υποσύνολο των χαρακτηριστικών που απαιτούνται από ένα κατανεμημένο σύστημα αποθήκευσης ώστε να είναι έτοιμο για χρήση σε περιβάλλοντα παραγωγής, και δεν έχει γίνει μέχρι στιγμής καμία προσπάθεια προσάρτησης σε αυτό κάποιας κατάλληλης μηχανής αποθήκευσης.

Αρχικές Υποθέσεις και Διαπροσωπεία

Το πρώτο βήμα στη σχεδίαση του κατανεμημένου συστήματος αποθήκευσης της εργασίας είναι η απόφαση σχετικά με το μοντέλο αποθήκευσης δεδομένων που θα χρησιμοποιηθεί για την ανάγνωση, εγγραφή και διαχείριση της πληροφορίας.

Το σύστημα σχεδιάζεται και υλοποιείται ως ένα σύστημα αποθήκευσης κλειδιού τιμής. Για την αποθήκευση και την διαχείρισή τους, τα δεδομένα έχουν τη μορφή ζευγών κλειδιού-τιμής ή αντικειμένων. Κάθε τμήμα πληροφορίας που πρέπει να αποθηκευτεί θεωρείται ένα μεγάλο δυαδικό αντικείμενο, δηλαδή ένα σύνολο δεδομένων σε δυαδική μορφή, το οποίο μπορεί να θεωρηθεί ως μία οντότητα. Με αυτή τη μορφή αποθηκεύεται και μόνιμα: ως μία ακολουθία ψηφιολέξεων. Κατά συνέπεια, για το σύστημά μας είναι εντελώς αδιάφορο το είδος της πληροφορίας που αποθηκεύεται, και το τι αυτό αναπαριστά για τους χρήστες, και ούτε απαιτείται ούτε υποστηρίζεται οποιουδήποτε είδους επιπρόσθετη μοντελοποίηση ή σχήμα για τα δεδομένα. Το μεγάλο δυαδικό αντικείμενο αντιστοιχεί στην "τιμή" του ζεύγους κλειδιού-τιμής. Κάθε φορά που ένα νέο αντικείμενο δημιουργείται για ένα εισερχόμενο μεγάλο δυαδικό αντικείμενο, το σύστημα αναθέτει ένα μοναδικό κλειδί σ᾽αυτό, ώστε να διαμορφωθεί το ζεύγος κλειδιού-τιμής. Το σύστημα ενημερώνει τους χρήστες γι᾽αυτό το κλειδί μέσω των απαντήσεών του στα αιτήματα δημιουργίας, διότι το κλειδί αποτελεί τον μοναδικό τρόπο

αναγνώρισης και προσπέλασης κάθε μεγάλου δυαδικού αντικειμένου στο μέλλον.

Τα αντικείμενα αποθηκεύονται σε έναν μοναδικό και επίπεδο χώρο ονομάτων. Δεν υπάρχουν πρόσθετοι "κουβάδες", "συλλογές" ή "περιέκτες" αντικειμένων, συνεπώς δεν υπάρχει τρόπος να επιβληθεί κάποια ιεραρχική – ή οποιουδήποτε άλλου είδους – δόμηση των δεδομένων. Τέτοιες ανάγκες, όποτε υπάρχουν, θεωρούμε ότι θα έπρεπε να καλύπτονται σε υψηλότερο επίπεδο στην αρχιτεκτονική κάποιου συστήματος που χρησιμοποιεί το παρόν ως στρώμα αποθήκευσης των δεδομένων του.

Σε αντίθεση με διάφορα συστήματα αποθήκευσης αντικειμένων, η τρέχουσα σχεδίασή μας δεν έχει λάβει υπ'όψιν την ανάγκη για αποθήκευση μεταδεδομένων για τα αντικείμενα. Η αποθήκευση μεταδεδομένων θεωρείται συνήθως ένα χρήσιμο χαρακτηριστικό, ιδίως για συστήματα αποθήκευσης αντικειμένων, τα οποία συχνά παρέχουν τη δυνατότητα καταχώρησης χρήσιμης πληροφορίας μέσω των μεταδεδομένων, συγκεκριμένης μορφής και περιεχομένου ανά εφαρμογή που τα χρησιμοποιεί. Παρ'ολ'αυτά, η αποθήκευση μεταδεδομένων με τρόπο ώστε να καλύπτει τα χαρακτηριστικά που επιδιώκουμε για το σύστημά μας είναι μία αρκετά σύνθετη διαδικασία, στην οποία δεν επιδιώκουμε να εστιάσουμε. Μία τέτοια προσεκτική υλοποίηση θα απαιτούσε την ενσωμάτωση επιπρόσθετων στιβάδων στην τρέχουσα προτεινόμενη αρχιτεκτονική, και συνεπώς φαίνεται να συμπνέει με τις σκέψεις μας για μελλοντικές επεκτάσεις της αρχιτεκτονικής του συστήματος.

Η υποδομή υλικού υπό του συστήματός μας, θεωρούμε ότι αποτελείται από πολλά φθηνά, "ευρέως εμπορευματοποιημένα" στοιχεία, τόσο για την ίδια την αποθήκευση (δηλαδή, για τους δίσκους), όσο και την επεξεργασία και δικτύωση. Αναμένεται ότι πολλά από αυτά τα στοιχεία θα αποτυγχάνουν, πολύ πιθανόν τις πλέον ακατάλληλες στιγμές. Το σύστημά μας πρέπει με κάποιον τρόπο να ανιχνεύει, να ανέχεται και να επανακάμπτει από τέτοιου είδους σφάλματα ως τμήμα της μόνιμης και φυσιολογικής του λειτουργίας. Αυτός είναι και ένας από τους βασικούς λόγους που χρησιμοποιείται το Kubernetes, σε συνδυασμό και με τεχνικές δημιουργίας αντιγράφων των δεδομένων, πράγματα τα οποία πρόκειται να συζητηθούν σύντομα.

Το σύστημα έχει σχεδιαστεί έχοντας ως στόχο την υποστήριξη συνολικής χωρητικότητας 1 PiB δεδομένων, εξαιρουμένων των αντιγράφων – όταν δηλαδή ο "παράγοντας αντιγραφής"[37] είναι 1, διαφορετικά η χωρητικότητα που υποστηρίζεται είναι

[37]Με τον ελληνικό όρο "παράγοντας αντιγραφής" αναφερόμαστε στην έννοια που εκφράζεται με

ακόμα μεγαλύτερη. Αποθηκεύει "μεγάλα δυαδικά αντικείμενα" σχετικά μικρού μεγέθους, της τάξης των μερικών MB το καθένα. θεωρώντας μεγάλα δυαδικά αντικείμενα περίπου των 4 MiB το καθένα κατά μέσο όρο, που είναι το παράδειγμα χρήσης που εστιάσαμε περισσότερο, το σύστημα πρέπει να είναι ικανό να αποθηκεύσει περίπου $2^{28} \approx 2.68 \times 10^8$ τέτοια αντικείμενα. Ωστόσο, τα "μεγάλα δυαδικά αντικείμενα" μπορεί, βεβαίως, να είναι μικρότερου μεγέθους, και συνεπώς το συνολικό τους πλήθος, μέχρι να καλύψουν συνολική χωρητικότητα 1 PiB, να είναι πολύ μεγαλύτερο. Λόγου χάριν, αν κάθε μεγάλο δυαδικό αντικείμενο είναι κατά μέσο όρο 4 KiB, το σύστημά μας θα πρέπει να είναι ικανό να διαχειριστεί περίπου $2^{38} \approx 2.75 \times 10^{11}$ τέτοια αντικείμενα, και προφανώς σε μερικές περιπτώσεις ακόμα περισσότερα.

Τα συστήματα αποθήκευσης κλειδιού-τιμής συνήθως παρέχουν απλές και λιτές διεπαφές, κρύβοντας τεχνικές λεπτομέρειες για την αποθήκευση και διαχείριση των δεδομένων από τις εφαρμογές-πελάτες τους. Αυτή είναι η λογική και της διεπαφής για το δικό μας σύστημα. Αφού ο κύριος στόχος της εργασίας είναι να εξερευνήσουμε περιεκτικοποιημένες μεθόδους αποθήκευσης πάνω απο το Kubernetes, δεν εστιάζουμε στην παροχή ευρείας ποικιλίας λειτουργιών πάνω στα αποθηκευμένα δεδομένα. Αντί αυτού, επιλέγουμε να υποστηρίξουμε μόνο ένα μικρό υποσύνολο των βασικών CRUD[38] πράξεων. Συγκεκριμένα, αναφερόμαστε μόνο στις λειτουργίες δημιουργίας και ανάγνωσης[39].

Η παντελής έλλειψη υποστήριξης πράξεων μεταβολής των δεδομένων στο σύστημά μας, άρα και πράξεων διαγραφής τους – αφού η διαγραφή σχεδόν πάντα θεωρείται ως μία ιδιάζουσα περίπτωση μεταβολής – πρέπει να τονιστεί. Πράξεις μεταβολής των δεδομένων εντός χρονικών διαστημάτων κατά τα οποία κάποιοι από τους κόμβους αποτυγχάνουν, συνιστούν μία από τις βασικότερες αιτίες ασυνέπειας των δεδομένων στα κατανεμημένα συστήματα αποθήκευσης, και έναν από τους κύριους λόγους για τους οποίους υπάρχουν αλγόριθμοι κατανεμημένης συμφωνίας, είτε απλοί,

---

τον αντίστοιχο αγγλικό "replication factor".

[38] Το ακρωνύμιο "CRUD" αναφέρεται στα "`Create`"-"Δημιουργία", "`Read`"-"Ανάγνωση", "`Update`"-"Μεταβολή", και "`Delete`"-"Διαγραφή", τα οποία είναι οι τέσσερις βασικές κατηγορίες πράξεων σε μονίμως αποθηκευμένα δεδομένα [94]. Μερικές φορές το ακρωνύμιο επεκτείνεται σε "CRUDL" ώστε να συμπεριλάβει την πράξη "`List`"-"Απαρίθμηση", η οποία μπορεί να αποτελέσει πρόκληση σε περιπτώσεις συνόλων δεδομένων που είναι πολύ μεγάλα για να χωρέσουν στη μνήμη του συστήματος.

[39] Η λειτουργία της απαρίθμησης, αν και μελετήθηκε διεξοδικά, εν τέλει υλοποιήθηκε μόνο σε πρώιμες μορφές του συστήματος

όπως οι 2PC[40] και 3PC[41] [95, 96], είτε πιο σύνθετοι, όπως η οικογένεια αλγορίθμων Paxos[119, 120, 121, 122, 123, 124], και οι Raft[125], Zab[126] και Viewstamp Replication[127]: όλοι τους έχουν στόχο να παρέχουν κάποιου είδους σύμπνοια μεταξύ μίας ομάδας οντοτήτων υπό το καθεστώς διαφορετικών, ιδιαίτερων συνθηκών και μοντέλων αποτυχιών. Μη επιτρέποντας τροποποιήσεις σε αποθηκευμένα δεδομένα, ουσιαστικά καταφέρνουμε να προσπεράσουμε τελείως αυτά τα εμπόδια, πάντα βέβαια με το αντίστοιχο κόστος: των εξαιρετικά μειωμένου πλήθους περιπτώσεων στις οποίες η χρήση του συστήματός μας δύναται να αποτελέσει αποδεκτή και βιώσιμη λύση. Επιπροσθέτως, αυτή η επιλογή μας επιτρέπει, όχι απλώς να εστιάσουμε στην αρχιτεκτονική του συστήματός πάνω από τα Kubernetes και Docker, αλλά και να εκμεταλλευτούμε τις έννοιες της αμεταβλητότητας και διευθυνσιοδότησης-βάσει-περιεχομένου, με τρόπο που θα περιγραφεί σύντομα.

Όπως είναι μάλλον ήδη πασιφανές, ο φόρτος εργασίας του συστήματος αποτελείται από αιτήματα πράξεων δημιουργίας και ανάγνωσης μεγάλων δυαδικών αντικειμένων. Ταυτοχρόνως ενεργούντα αιτήματα πελατών, τα οποία πιθανώς αναφέρονται σε κοινά μεγάλα δυαδικά αντικείμενα, πρέπει να μπορούν να ικανοποιούνται παραλλήλως από το σύστημα, και συνεπώς η μηχανή αποθήκευσής του πρέπει να είναι σε θέση να τα διαχειριστεί κατάλληλα.

Επικοινωνία

Σύμφωνα με τις βασικές αρχές σχεδίασης εφαρμογών και συστημάτων υπηρεσιών Cloud, οι πελάτες αποστέλλουν τα αιτήματά τους και λαμβάνουν τις αντίστοιχες απαντήσεις πάντα μέσω του δικτύου. Το ίδιο κάνουν και οι ίδιες οι οντότητες που απαρτίζουν το σύστημα για την εσωτερική τους επικοινωνία.

Για κάθε είδους δικτυακής επικοινωνίας, χρησιμοποιούμε την γνωστή στοίβα πρωτοκόλων TCP/IP, όπως είναι υλοποιημένη στον πυρήνα του Linux. Η διαδικασία της επιλογής αυτής ήταν τετριμμένη. Το πρωτόκολλο IP είναι de facto καθιερωμένο στο επίπεδο δικτύου. Στο επίπεδο μεταφοράς, τα δύο επικρατέστερα υποψήφια πρωτόκολλα είναι τα TCP και UDP. Η πιο θεμελιώδης ευθύνη των πρωτοκόλλων επιπέδου

[40] Πρωτόκολλο "Two-Phase Commit".
[41] Πρωτόκολλο "Three-Phase Commit".

μεταφοράς είναι να επεκτείνουν την υπηρεσία διανομής μεταξύ δύο συστημάτων που παρέχεται από το IP, σε υπηρεσία διανομής μεταξύ δύο διεργασιών του λειτουργικού συστήματος οι οποίες εκτελούνται στα δύο αυτά συστήματα. Αυτή η διαδικασία ονομάζεται πολύπλεξη και αποπολύπλεξη επιπέδου μεταφοράς [97].

Αφενός, το UDP είναι ένα ασυνδεσμικό πρωτόκολλο βασισμένο σε δεδομενογράμματα, παρέχει τις ελάχιστες υπηρεσίες που απαιτούνται ώστε να μπορεί να σταχυολογηθεί ως πρωτόκολλο επιπέδου μεταφοράς. Αυτές είναι ουσιαστικά η πολύπλεξη και αποπολύπλεξη επιπέδου μεταφοράς, παρέχει όμως και έλεγχο ακεραιότητας των δεδομένων συμπερικλείοντας πεδία ανίχνευσης λαθών στην επικεφαλίδα κάθε τμήματος. Όπως και το IP, το UDP παρέχει γενικά αναξιόπιστη υπηρεσία, αφού δεν εγγυάται ούτε ότι τα δεδομένα που αποστέλλονται από μία διεργασία θα φτάσουν ορθώς, ούτε ότι θα φτάσουν καν, στην διεργασία-παραλήπτη [97]. Γι᾽αυτόν τον λόγο, το UDP είναι δημοφιλής επιλογή για εφαρμογές που είτε απαιτούν πολύ ακριβή έλεγχο και λεπτεπίλεπτο χειρισμό ως προς το πώς και το πότε στέλνεται η πληροφορία στο επίπεδο της εφαρμογής, είτε μπορούν να ανεχτούν απώλειες πληροφορίας προς όφελος της αποδοτικότητας και της ταχύτητας διανομής, σε βάρος της αξιοπιστίας.

Αφετέρου, το TCP είναι ένα συνδεσμικό πρωτόκολλο, που πρωτίστως παρέχει αξιόπιστη μεταφορά δεδομένων πάνω από το αναξιόπιστο πρωτόκολλο IP. Γι᾽αυτόν τον σκοπό, έχει χρησιμοποιεί ένα πλήθος "ευφυών" τεχνικών, συμπεριλαμβανομένων των: ελέγχο ροής, αριθμοί ακολουθίας, επιβεβαιώσεις, χρονομετρητές, και ακόμα και μηχανισμό ελέγχου συμφόρησης ώστε να αποτρέψει μεμονωμένες συνδέσεις TCP να πλημμυρίσουν τις ζεύξεις και τους δρομολογητές μεταξύ των κόμβων που επικοινωνούν. [97]. Για τον λόγο αυτόν, το TCP είναι πιο σύνθετο στην υλοποίησή του, ακόμα και στην ρύθμισή του, αλλά συνιστά και την συνήθη επιλογή για εφαρμογές που έχουν ανάγκη να μεταφέρουν δεδομένα σωστά και με σειρά.

Το πολύ μεγάλο πλήθος ρυθμιζόμενων χαρακτηριστικών του TCP παρέχει εν δυνάμει επιπρόσθετα ωφέλη σε εφαρμογές και συστήματα που το χρησιμοποιούν. Τέτοιες εφαρμογές και συστήματα μπορούν να ρυθμίσουν ποικίλες παραμέτρους του TCP και να υλοποιήσουν εξατομικευμένα πρωτόκολλα επιπέδου εφαρμογής για μεταφορά δεδομένων πάνω από τη στοίβα TCP/IP. Παρ᾽ολ᾽αυτά, ούτε η σχεδίαση και υλοποίηση αποδοτικών πρωτοκόλλων μεταφοράς δεδομένων επιπέδου εφαρμογής, ούτε και η τέλεια ρύθμιση όλων των παραμέτρων του πρωτοκόλλου TCP είναι πάντα τετριμμένη

διαδικασία, και σίγουρα είναι και οι δύο εκτός των στόχων της παρούσας εργασίας. Συνεπώς, αποφασίσαμε να προσκολληθούμε σε μία από τις βασικές πρακτικές της σχεδίασης σύμφωνα με την Αρχιτεκτονική Μικροϋπηρεσιών: να χρησιμοποιήσουμε το πρωτόκολλο HTTP ως πρωτόκολλο επιπέδου εφαρμογής.

Το HTTP είναι το "πρωτόκολλο επιπέδου εφαρμογής του Παγκόσμιου Ιστού" και βασίζεται στο μοντέλο πελάτη-εξυπηρέτη. Ένας εξυπηρέτης λαμβάνει αιτήματα πελατών, και στη συνέχεια παράγει και στέλνει την κατάλληλη, κάθε φορά, απάντηση. Το HTTP είναι ένα πρωτόκολλο που δεν διατηρεί μόνιμη κατάσταση, με την έννοια ότι οι εξυπηρέτες δημιουργούν απαντήσεις για τα εισερχόμενα αιτήματα χωρίς να αποθηκεύουν οποιουδήποτε είδους πληροφορία κατάστασης των πελατών. Στο επίπεδο μεταφοράς του, προϋποθέτει ένα αξιόπιστο πρωτόκολλο, και συνεπώς το TCP είναι συνήθως, και εν προκειμένω, η επιλογή. Η αρχική πρόταση του πρωτοκόλλου είναι το `HTTP/1.0` [98], το οποίο εκ των υστέρων αναθεωρήθηκε με την έκδοση του πρωτοκόλλου `HTTP/1.1` [99]. Σήμερα, ακόμα και το τελευταίο αντικαθίσταται σταδιακά από το πρωτόκολλο `HTTP/2` [100], το οποίο προέρχεται, έχοντας διατηρήσει πάρα πολλά κοινά στοιχεία, από το πρωτόκολλο `SPDY` [101], το οποίο έχει αναπτυχθεί από την εταιρεία Google.

Η εξωτερική Διεπαφή Προγραμματισμού Εφαρμογών του συστήματός μας είναι σχεδιασμένο έχοντας σαν βάση το πρωτόκολλο HTTP, και δύο διαφορετικά υποδείγματα: το REST[42] και το RPC[43]. Παρότι και τα δύο αυτά υποδείγματα μελετήθηκαν στο πλαίσιο της παρούσας εργασίας, πρέπει να σημειώσουμε ότι η πρωτόγονη λιτότητα της Διεπαφής Προγραμματισμού Εφαρμογών του συστήματός μας δεν επέτρεψε σε κανένα εκ των δύο να ξεδιπλώσει όλα τα πλεονεκτήματα και τα μειονεκτήματά του, και άρα ούτε μία πλήρη και ουσιαστική σύγκριση μεταξύ των δύο μεθόδων. Ωστόσο, για λόγους που σχετίζονται με την υλοποίηση, η μέθοδος RPC αποδείχθηκε καλύτερη σε ζητήματα αποδοτικότητας, και γι᾽αυτό και επιλέχθηκε στην τελική υλοποίηση της Διεπαφής Προγραμματισμού Εφαρμογών του συστήματος.

[42] Ακρωνύμιο για το "Representational State Transfer".
[43] Ακρωνύμιο για το "Remote Procedure Call".

REST, JSON & Base64

Το "REST" [102] είναι μία μέθοδος αρχιτεκτονικής που εφαρμόζεται στη σχεδίαση υπηρεσιών ιστού και δικτυακές εφαρμογές. Υπηρεσίες που συμμορφώνονται με αυτό, εκθέτουν την κατάσταση και την λειτουργικότητά τους ως ένα σύνολο πόρων, και επιτρέπουν τα αιτούντα συστήματα να τα προσπελάσουν και να τα διαχειριστούν μέσω ενός προκαθορισμένου συνόλου πράξεων οι οποίες δεν έχουν μόνιμη κατάσταση. Όταν συνδυάζεται με το πρωτόκολλο HTTP, αυτές οι πράξεις αντιστοιχίζονται σε "ρήματα" του HTTP, όπως τα `GET`, `PUT`, `POST`, `DELETE`, και άλλα. Όλοι αυτοί οι πόροι είναι προσπελάσιμοι με μοναδικό τρόπο, συνήθως μέσω URI[44], και τα σχετικά δεδομένα τους μεταφέρονται αναπαριστώμενα με κάποια γνωστή μορφή, όπως HTML, XML, JSON, και λοιπές.

Η "πρωτόγονη" REST Διεπαφή Προγραμματισμού Εφαρμογών του συστήματός μας είναι σχεδιασμένη και υλοποιημένη χρησιμοποιώντας το πρωτόκολλο `HTTP/1.1`, χρησιμοποιώντας JSON[45][103] ως μορφή αναπαράστασης μεταφερόμενων δεδομένων. Το JSON από τη φύση του δεν υποστηρίζει δυαδικά δεδομένα. Για την αξιόπιστη μεταφορά δυαδικών δεδομένων με μέσα που είναι σχεδιασμένα για να διαχειρίζονται δεδομένα σε μορφή κειμένου, είναι αναγκαίο να προηγείται η κωδικοποίησή τους. Αυτό γεννά το ζήτημα της επιλογής ενός αποδοτικού αλγορίθμου κωδικοποίησης δεδομένων μορφής δυαδικής-σε-κειμένου, και ακολούθως της εξερεύνησης των πλεονεκτημάτων και μειονεκτημάτων τους, κυρίως σε ό,τι αφορά την αποδοτικότητά τους από τη σκοπιά των απαιτούμενων υπολογισμών και του μεγέθους της κωδικοποιημένης αναπαράστασης που παράγεται.

Επιλέχθηκε ο αλγόριθμος Base64[104] για την κωδικοποίηση από δυαδική μορφή σε μορφή κειμένου. Είναι δημοφιλής μεταξύ εφαρμογών που ακολουθούν το REST υπόδειγμα και αναπαράσταση JSON σε συνδυασμό με το πρωτόκολλο HTTP για τις Διεπαφές Προγραμματισμού Εφαρμογών τους, όταν έχουν την ανάγκη να μεταφέρουν δυαδικά δεδομένα, και έτσι υπάρχουν πολλές έτοιμες και ποιοτικές υλοποιήσεις του, από τις οποίες μπορούμε να επιλέξουμε. Κάθε ψηφίο της εξόδου Base64 αναπαριστά ακριβώς 6 δυφία δεδομένων – με άλλα λόγια, κάθε ομάδα τριών (οκταδύφιων) ψηφιολέξεων εισόδου αναπαρίσταται με τέσσερα ψηφία Base64. Επομένως, η Base64 ανα-

[44] Ακρωνύμιο για "Uniform Resource Identifier".
[45] Ακρωνύμιο για το "JavaScript Object Notation".

παράσταση μίας ακολουθίας ψηφιολέξεων είναι πάντα τουλάχιστον 25% μεγαλύτερη από την αρχική της. Αφού η κωδικοποίηση, και η αποκωδικοποίηση, λαμβάνει χώρα αμέσως πριν την απόστολη, και αμέσως μετά την παραλαβή, αντιστοίχως, κάθε μηνύματος HTTP, εκτός του πρόσθετου υπολογιστικού κόστους τους, υπάρχει και το μη αμελητέο κόστος της περίπου 25% επιπρόσθετης χρησιμοποίησης εύρους ζώνης δικτύου για την μεταφορά καθαυτή.

RPC, Protocol Buffers & gRPC

Μία εναλλακτική έννοια για τη σχεδίαση υπηρεσιών ιστού και δικτυακών εφαρμογών είναι αυτή των "απομακρυσμένων κλήσεων διαδικασιών" ("Remote Procedure Calls" ή "RPC") [105]. Πρόκειται ουσιαστικά για μια επέκταση της ευρέως γνωστής και μελετημένης αφαίρεσης της κλήσης διαδικασίας, σε κατανεμημένα περιβάλλοντα [95]. Όταν πραγματοποιείται μία κλήση RPC, το "καλόν περιβάλλον" παύεται και οι παράμετροι περνώνται μέσω του δικτύου στο "καλούμενο περιβάλλον", δηλαδή εκεί που θα εκτελεστεί η διαδικασία, μέχρι η επιθυμητή διαδικασία να εκτελεστεί και τα παραχθέντα αποτελέσματα να αποσταλούν στο καλόν περιβάλλον, οπότε και η εκτέλεσή του συνεχίζεται, σαν να επέστρεφε από μία απλή κλήση διαδικασίας τοπικά [105]. Πρόκειται, συνεπώς, ουσιαστικά για μία μορφή αλληλεπίδρασης πελάτη-εξυπηρέτη και ένα είδος πρωτοκόλλου αιτημάτων-απαντήσεων, όπου ο πελάτης είναι ο καλών, ο οποίος εκκινεί τη διαδικασία μέσω μίας τοπικής του ψευδο-συνάρτησής, και ο εξυπηρέτης είναι ο καλούμενος, ο οποίος παράγει την κατάλληλη απάντηση και απαντάει μέσω μίας δικής του τοπικής ψευδο-συνάρτησης.

Το RPC μπορεί να χρησιμοποιηθεί ως κεντρική έννοια στη σχεδίαση της Διεπαφής Προγραμματισμού Εφαρμογών του συστήματός μας, ως υποκατάστατο του REST. Ασφαλώς, όλα τα προβλήματα που ταλανίζουν την ίδια τη μεταφορά των δεδομένων δεν παύουν να υφίστανται. Τα δεδομένα ακόμα χρειάζεται να σειριοποιηθούν και να αποσειριοποιηθούν για τη μεταφορά, και αφού μπορεί να είναι δυαδικά, πρέπει επίσης να κωδικοποιούνται και να αποκωδικοποιούνται ώστε να αποφευχθούν απώλειες χρήσιμης πληροφορίας. Στη συνέχεια, πρέπει να αποσταλούν στο δίκτυο χρησιμοποιώντας κάποιο πρωτόκολλο επιπέδου εφαρμογής. Και όλα αυτά πρέπει να γίνουν αποδοτικά. Θα περιγράψουμε, λοιπόν, μία σύγχρονη και αποδοτική μορφή σειριοποίησης, εναλλακτικής του JSON, και μία βιβλιοθήκη RPC που το χρησιμοποιεί για να

επιτύχει υψηλές επιδόσεις, χωρίς να θυσιάζει την απλότητα και την ευκολία χρήσης του.

Το "protocol buffers" ή "protobuff" είναι μία επεκτάσιμη, ανεξάρτητη γλώσσας και πλατφόρμας μέθοδος σειριοποίησης δομημένων δεδομένων για χρήση κυρίως σε πρωτόκολλα επικοινωνίας και αποθήκευση δεδομένων. Αναπτύχθηκε αρχικά από την εταιρεία Google για εσωτερική χρήση, αλλά αργότερα δημοσιεύτηκε ως βιβλιοθήκη ανοικτού κώδικα [106].

Το "protocol buffers" είναι ευπροσάρμοστο και αποδοτικό, και συνοδεύεται από το εργαλείο `protoc`, το οποίο παρέχει την σε μεγάλο βαθμό αυτοματοποίηση πολλών σχετικών διαδικασιών. Με το "protocol buffers", ο προγραμματιστής αρκεί να ορίσει τις δομές δεδομένων (τα οποία στο παρόν πλαίσιο ονομάζονται και "μηνύματα") σε ένα αρχείο `.proto`, χρησιμοποιώντας μία Γλώσσα Ορισμού Διαπροσωπείας[46], η οποία προς το παρόν υπάρχει σε δύο εκδόσεις, τις `proto2` και `proto3`. Βάσει αυτών των ορισμών, ο μεταγλωττιστής του "protocol buffers" παράγει με αυτόματο τρόπο πηγαίο κώδικα, έτοιμο για χρήση από την εφαρμογή: δομές δεδομένων και συναρτήσεις που υλοποιούν την κωδικοποίηση και αποκωδικοποίηση των δεδομένων. Μέχρι στιγμής (Μάιος 2018), το "protocol buffers" υποστηρίζει τις εξής γλώσσες προγραμματισμου: Go, Python, Java, C++, C#, Objective-C, Javascript, Ruby, PHP και Dart, αν και σε μερικές περιπτώσεις υπάρχουν μικρά ζητήματα ασυμβατότητας μεταξύ κάποιων εξ αυτών.

Η αποδοτικότητα των "protocol buffers" πηγάζει, όχι μόνο από την εξαιρετική υποστήριξη αυτοματοποίησης, αλλά και από το ίδιο το σχήμα της κωδικοποίησης. Τα μηνύματα του "protocol buffer" σειριοποιούνται σε μία συμπεπτυγμένη, γρήγορα επεξεργάσιμη, δυαδική μορφή, η οποία όμως δεν είναι αυτο-περιγραφόμενη (δηλαδή, αν δεν είναι γνωστό το αντίστοιχο αρχείο `.proto`, δεν υπάρχει τρόπος να αποκωδικοποιηθεί αυτή η δυαδική μορφή, σε αντίθεση με περιπτώσεις όπως το JSON και το XML, τα οποία, ως ένα βαθμό, το επιτρέπουν). Για περιπτώσεις όπως η δική μας, όπου τα μεγάλα δυαδικά αντικείμενα πρέπει να μεταφερθούν μέσω του δικτύου, τα "protocol buffers" είναι μία σημαντική βελτίωση σε σχέση με τον συνδυασμό Base64 και JSON, αναφορικά τόσο με τις επιδόσεις της σειριοποίησης και αποσειριοποίησης, όσο και με

[46]Με τον όρο "Γλώσσα Ορισμού Διαπροσωπείας" αναφερόμαστε στην έννοια γνωστή με την αγγλική ορολογία "Interface Definition/Description Language" ή "IDL".

το χρησιμοποιούμενο εύρος ζώνης του δικτύου που χρησιμοποιείται για τη μεταφορά των δεδομένων.

Το gRPC είναι μία βιβλιοθήκη RPC, η οποία αναπτύχθηκε αρχικά από την εταιρεία Google ως μία επανέκδοση του εσωτερικού RPC συστήματός της, ονόματι Stubby, και το οποίο αργότερα δημοσιεύτηκε ως λογισμικό ανοικτού κώδικα.

Όπως αρκετές βιβλιοθήκες RPC, κεντρική ιδέα του είναι ο ορισμός των υπηρεσιών και μεθόδων που μπορούν να καλούνται απομακρυσμένα, με τις παραμέτρους τους και τους τύπους επιστροφής τους. Ο διακομιστής gRPC υλοποιεί αυτή τη διαπροσωπεία και χειρίζεται τις κλήσεις πελατών, ενώ στους πελάτες gRPC παρέχεται ένα σύνολο ψευδο-συναρτήσεων, παρόμοιων με αυτές που ορίζονται στον αντίστοιχο διακομιστή. Ο διακομιστής και οι πελάτες μπορεί να υλοποιηθούν σε διάφορες γλώσσες προγραμματισμού, και όχι αναγκαστικά στις ίδιες μεταξύ τους. Προς το παρόν υποστηρίζονται οι εξής: Go, Python, Java, C++, C#, Objective-C, Node.js, Ruby, PHP και Dart.

Το gRPC μπορεί ουσιαστικά να θεωρηθεί ως ένα πρωτόκολλο σε ένα επίπεδο πάνω από το `HTTP/2` [100]. Η προεπιλεγμένη μορφή μεταφοράς των δεδομένων, με την οποία υπάρχει και εγγύηση για υψηλή αποδοτικότητα, είναι το "protocol buffers". Αυτός ο συνδυασμός έχει δοκιμαστεί με επιτυχία και σε πληθώρα περιπτώσεων σε περιβάλλοντα παραγωγής. Η Γλώσσα Ορισμού Διαπροσωπείας του "protocol buffers' χρησιμοποιείται τόσο για τον ορισμό των δομών δεδομένων που μεταφέρονται μεταξύ πελατών και διακομιστών, όσο και για τις ίδιες τις παρεχόμενες gRPC υπηρεσίες και οι διαδικασίες που αυτές εκθέτουν. Ύστερα, το εργαλείο `protoc` χρησιμοποιείται για την παραγωγή πηγαίου κώδικα βάσει του αρχείου `.proto` που περιέχει τους παραπάνω ορισμούς. Αυτή τη φορά, όμως, χρησιμοποιείται μαζί με ένα ειδικό πρόσθετο εργαλείο διασύνδεσής του με το gRPC, το οποίο, εκτός του συνήθους κώδικα του "protocol buffers", παράγει επιπρόσθετο πηγαίο κώδικα, έτοιμο για χρήση από διακομιστές και πελάτες gRPC.

Αμεταβλητότητα

Θεμελιωδώς, η αμεταβλητότητα εκφράζει την απλή ιδέα ότι τα δεδομένα παραμένουν αμετάβλητα μετά τη δημιουργία τους.

Ως έννοια, δεν είναι καινοτόμα. Οι ρίζες της εντοπίζονται στο συναρτησιακό υπόδειγμα προγραμματισμού, και συνεπώς έχει ισχυρή παρουσία σε γλώσσες όπως οι Haskell, Erlang και ML. Απαντάται βέβαια και σε άλλες γλώσσες προγραμματισμού, δημοφιλέστερες σε περιβάλλοντα παραγωγής, όπως οι Java, Python και Go, οι οποίες, επί παραδείγματι, χειρίζονται τις συμβολοσειρές ως μη μεταβαλλόμενα δεδομένα. "Μόνιμες δομές δεδομένων" (δομές δεδομένων οι οποίοες πάντα διατηρούν προηγούμενες εκδόσεις των εαυτών τους όποτε τροποποιούνται[109], επιτρέποντας έτσι ερωτήματα και τροποποιήσεις σε όλες τις προηγούμενες εκδόσεις τους, οι οποίες τελικώς διαμορφώνουν ένα δέντρο εκδόσεων), οι οποιες χρησιμοποιούνται ευρύτατα στον παράλληλο προγραμματισμό για την ευκολία που παρέχουν, συνήθως υλοποιούνται βάσει της έννοιας της αμεταβλητότητας.

Η αμεταβλητότητα έχει αποτελέσει και τη βάση για "Συστήματα Ελέγχου Εκδόσεων"[47]. Λόγου χάριν, το μοντέλο του Git[110] θυμίζει μία δομή δεδομένων συναρτησιακού υποδείγματος η οποία λειτουργεί στο δίσκο, και έχει μία Διεπαφή Γραμμής Εντολών για την εκτέλεση πράξεων πάνω σε αυτή [111]. Στον πυρήνα του, το Git περιλαμβάνει "commits", δηλαδή ανεξάρτητα πλήρη στιγμιότυπα του καταλόγου υπό επεξεργασία, τα οποία διαμορφώνουν μία "ιστορία", δηλαδή ένα δέντρο εκδόσεων, στο οποίο μπορούν να γίνουν αναφορές σε μεμονωμένα "commits" μέσω "κλάδων" ή "branches", δηλαδή δείκτες σε αυτά τα "commits". Στην πραγματικότητα, το Git υποστηρίζει και μεταβολές, όπως παραδείγματος χάριν την επανεγγραφή της "ιστορίας", όμως σαν πρακτική είναι αμφιλεγόμενη και θεωρείται αποδεκτή μόνο υπό προϋποθέσεις.

Η έλευσης της εποχής της "ευρείας εμπορευματοποίησης" των υποδομών υλικού στα κέντρα δεδομένων, έχει επιφέρει την εισαγωγή της έννοιας της αμεταβλητότητας και σε ζητήματα σχετικά με την αποθήκευση και τις βάσεις δεδομένων. Παραδοσιακά συστήματα διαχείρισης βάσεων δεδομένων δημιουργήθηκαν για πρώτη φορά σε μία εποχή που τα δεδομένα ήταν λίγα, ενώ τα μέσα αποθήκευσης ήταν ακριβά. Η προφανής λύση σε αυτό το πρόβλημα ήταν να κρατώνται τα δεδομένα σε εγγραφές, ανεξάρτητα του μοντέλου προσπέλασής τους, οι οποίες να μεταλλάσσονται συνεχώς μέσω δοσοληψιών – ένα μοντέλο που ανθεί ακόμα ύστερα από αρκετές δεκαετίες. Η αποδοτική διαχείριση των δοσοληψιών ήταν πάντα ένα θέμα σπουδαίου ερευνητικού ενδιαφέροντος, από ακαδημαϊκή και βιομηχανική άποψη. Προκειμένου να αποκομιστούν

[47] Με τον όρο "Σύστημα Ελέγχου Εκδόσεων" αναφερόμαστε στην έννοια που εκφράζει ο αντίστοιχος αγγλικός όρος "Version Control System".

βέλτιστες επιδόσεις, οι δοσοληψίες πρέπει να εκτελούνται παράλληλα. Επομένως, η παρουσία "έξυπνων" τεχνικών για τον συντονισμό τους είναι μια ανάγκη, και πράγματι, αρκετές τέτοιες έχουν μηχανορραφηθεί όλες αυτές τις δεκαετίες. Ανεξαρτήτως του αν αυτές είναι "αισιόδοξες" ή "απαισιόδοξες", δηλαδή αν χρησιμοποιούν "κλειδώματα" ή όχι, όλες τους στηρίζονται σε "χαμηλού επιπέδου κλειδώματα" τα οποία παρέχουν λεπτεπίλεπτο έλεγχο εσωτερικών δομών δεδομένων του συστήματος διαχείρισης βάσεων δεδομένων, όπως, παραδείγματος χάριν, σημαφόροι. Η χρήση "χαμηλού επιπέδου κλειδωμάτων" ήταν πάντα η λιγότερο ακριβή λύση.

Σήμερα, οι υπολογισμοί αλλά και η μνήμη γίνονται διαρκώς φθηνότεροι, χάρη στους πολυπύρηνους επεξεργαστές, τα σύγχρονα ολοκληρωμένα κυκλώματα και τους δίσκους στερεάς κατάστασης. Ο συντονισμός των νημάτων ενός συστήματος διαχείρισης βάσεων δεδομένων χρησιμοποιώντας "χαμηλού επιπέδου κλειδώματα" καθίσταται συνεχώς πιο επιβαρυντικός εξαιτίας του επαυξημένου χρόνου αναμονής και των ευκαιριών για περισσότερους υπολογισμούς που συνεπακολούθως χάνονται. Εντωμεταξύ, ο όγκος των δεδομένων καθαυτός, αλλά και το πλήθος των πελατών που αιτούνται αναγνώσεις και εγγραφές τους, έχουν αυξηθεί σε άλλωτε ασύλληπτα επίπεδα, χάρη στις εκρήξεις τεχνολογιών σχετικών με "Big Data", "Διαδίκτυο των Αντικειμένων", και λοιπά. Αναλόγως έχει αυξηθεί και η ανάγκη των σύγχρονων επιχειρήσεων να το αντιμετωπίσουν: τα δεδομένα πρέπει να αποθηκεύονται μόνιμα και με ασφάλεια, και να προσπελάζονται και να αναλύονται αξιόπιστα και αποδοτικά. Η διατήρηση αμετάβλητων αντιγράφων των δεδομένων είναι σήμερα μια οικονομικά βιώσιμη λύση, και θα μπορούσε συχνά να βοηθά στην ελαχιστοποίηση των καθυστερήσεων που προκαλούνται από τις "αντικαταστάσεις"[48], μειώνοντας έτσι τις προκλήσεις συντονισμού που αναφύονται στα σύγχρονα συστήματα αποθήκευσης [108].

Ο Martin Kleppmann χαρακτηρίζει τα παραδοσιακά συστήματα βάσεων δεδομένων ως "καθολική, διαμοιραζόμενη μεταβαλλόμενη κατάσταση" [144]. Θεωρεί ότι το γεγονος πως αυτή η κατάσταση δεν έχει αλλάξει σημαντικά από τη δεκαετία του 1960 είναι ανυπόφορο, ιδίως δεδομένου ότι η πρακτική της απομάκρυνσης αντίστοιχων καθολικών μεταβλητών σε πηγαίο κώδικα είναι ήδη ευρέως καθιερωμένη – και ορθώς.

Ένα πλήθος συστημάτων κάνουν πράγματι ένα βήμα παραπάνω προς την αμεταβλη-

[48] Με τον ελληνικό όρο "αντικατάσταση" εν προκειμένω αναφερόμαστε στην έννοια που εκφράζει ο αγγλικός όρος "overwriting".

τότητα. Ένα από τα γνωστότερα εξ αυτών είναι το Datomic[145], το οποίο παρέχει υψηλή διαθεσιμότητα μέσω της αυτοματοποιημένης δημιουργίας αντιγράφων των δεδομένων, και επαναφορά από σφάλματα, ισχυρή συνέπεια, ευπροσάρμοστο σχήμα δεδομένων, αποδοτική ευρετηριοποίηση, και υποστηρίζει ερωτήματα τόσο στην παρούσα όσο και σε παρελθούσες καταστάσεις των δεδομένων (ακόμα και σε υποθετικές μελλοντικές τέτοιες καταστάσεις) μέσω μίας δηλωτικής, λογικής γλώσσας ερωτημάτων. Ο Rich Hikey, ένας από τους σχεδιαστές του, ισχυρίζεται ότι η παρελθούσα πληροφορία είναι πάντα αμετάβλητη από τη φύση της: δεν μπορεί να αλλάξει. Συνεπώς, το μοντέλο πληροφορίας του Datomic βασίζεται στην επικάθηση αμετάβλητων ατομικών "γεγονότων" τα οποία στο πλαίσιο του Datomic εκφράζονται με τον όρο "datoms", τα οποία μπορεί είτε να προσθέτουν είτε να ανακαλούν πληροφορία ώστε να τροποποιήσουν την κατάσταση της βάσης δεδομένων (έτσι πραγματοποιούνται και τα ερωτήματα σε μελλοντικές καταστάσεις της βάσης: έναντι προσωρινών γεγονότων που αποθηκεύονται στη μνήμη και επικάθονται στα ήδη αποθηκευμένα). Είναι αξιοσημείωτο ότι το Datomic αποσκοπεί μόνο στην επιβολή μιας συγκεκριμένης αρχιτεκτονικής: το πραγματικό στρώμα αποθήκευσης είναι ένα "μαύρο κουτί" για το Datomic, και μπορεί να είναι οποιοδήποτε σύστημα αποθήκευσης υποστηρίζει σημασιολογικά χαρακτηριστικά τύπου κλειδιού-τιμής και συνεπείς αναγνώσεις [146]. Άλλα συστήματα που εκμεταλλεύονται την έννοια της αμεταβλητότητας και τα πλεονεκτήματά της με τον έναν ή τον άλλον τρόπο, είναι τα Tango[147] της Microsoft, Orchestrate[148] της CenturyLink, Apache CouchDB[149] και το RethinkDB[150], ιδίως στις πρώτες του εκδόσεις.

Ασφαλώς, όπως πολλές έννοιες και τεχνικές της τεχνολογίας λογισμικού, η δόμηση συστημάτων αποθήκευσης βάσει της αμεταβλητότητας έχει και τα μειονεκτήματά της. Μία τέτοια συνοπτική και περιεκτική κριτική εκφράζεται από τον Baron Schwartz στο [152], η οποία αναπτύσσεται χρησιμοποιώντας τα Datomic, RethinkDB και CouchDB ως παραδείγματα. Σύμφωνα μ᾽αυτή, κατ᾽αρχάς, η ικανότητα πρόσβασης σε παρελθούσες καταστάσεις και γεγονότα της βάσης δεδομένων μπορεί να αποβεί ιδιαιτέρως δαπανηρή όσον αφορά τον χρησιμοποιούμενο χώρο, ο οποίος θεωρητικά συνεχίζει να αυξάνεται επ᾽άπειρον, και ειδικά αναλογιζόμενοι ότι η πρόσβαση στο παρελθόν δεν είναι ένα χαρακτηριστικό που απαιτείται από την πλειοψηφία των σύγχρονων μοντέλων εφαρμογών. Το δεύτερο ζήτημα είναι η κατακερματισμένη αποθήκευση: καθώς νέα, αμετάβλητα γεγονότα επικάθονται διαρκώς, μία οντότητα συχνά καταλήγει δια-

σκορπισμένη σε πολλά σημεία του αποθηκευτικού μέσου, το οποίο μπορεί σε βάθος χρόνου να αποβεί μοιραίο για τις επιδόσεις του συστήματος, ακόμα και για δίσκους στερεάς κατάστασεις. Η δε επίλυση αυτού του ζητήματος συνήθως επιφέρει μεγάλη πολυπλοκότητα τόσο σε επίπεδο σχεδίασης, όσο και σε επίπεδο υλοποίησης του συστήματος. Επιπροσθέτως, στην κριτική αναφέρονται ποικίλοι περιορισμοί που τίθενται στην αρχιτεκτονική και την υλοποίηση των συστημάτων λόγω της υποστήριξης "μερικής αμεταβλητότητας", παραθέτοντας ως παράδειγμα τον τρόπο διαχείρισης του προβλήματος της πλήρωσης του δίσκου από το CouchDB. Τέλος, παρουσιάζεται ο ισχυρισμός ότι η χρήση "μερικής αμεταβλητότητας" σε παραδοσιακά συστήματα βάσεων δεδομένων, μέσω της τεχνικής MVCC[49] για την υποστήριξη ACID[50] δοσοληψιών είναι αποτέλεσμα πολλής σκέψης και πολλών αποπειρών σχεδίασης αρχιτεκτονικών συστημάτων βάσεων δεδομένων, και έχει ωριμάσει αρκετά, σε βάθος χρόνου δεκαετιών, ώστε να εμπνέει εμπιστοσύνη.

Όλες οι σχεδιαστικές μας προσεγγίσεις στοχεύουν στην πλήρη εκμετάλλευση της έννοιας της αμεταβλητότητας – τόσο όσο την εκμεταλλεύεται και το Datomic, παρότι εν προκειμένω το μοντέλο πρόσβασης και αποθήκευσης των δεδομένων, και οι σχετικές του λειτουργίες, είναι σε πολύ πρώιμο και απλούστερο από ότι του Datomic επίπεδο. Το στοιχείο της αρχιτεκτονικής μας που είναι υπεύθυνο για την αποθήκευση των δεδομένων, στην πραγματικότητα μία μικροϋπηρεσία, το επιτυγχάνει διατηρώντας τα εντελώς αμετάβλητα. Τα δεδομένα φτάνουν σ'αυτό στη μορφή ζευγών κλειδιού-τιμής, και αποθηκεύονται μόνιμα με αυτόν τον ακριβώς τον τρόπο. Τόσο τα κλειδιά όσο και οι τιμές τους είναι αμετάβλητες ακολουθίες ψηφιολέξεων. Τα κλειδιά είναι σταθερού μήκους, συνήθως όχι μεγαλύτερης τάξεως από 200 ψηφιολέξεις. Οι τιμές είναι μεγάλα δυαδικά αντικείμενα, ποικίλων μεγεθών μέχρι λίγα MB έκαστο. Το σύστημα είναι σχεδιασμένο ώστε να λειτουργεί για συνολική χωρητικότητα 1 PiB μοναδικών δεδομένων, με την έννοια ότι όταν είναι ενεργή η αυτοματοποιημένη δημιουργία αντιγράφων των δεδομένων, αυτό το όριο μπορεί να ξεπεραστεί χωρίς προβλήματα συμπεριλαμβάνοντας τα αντίγραφα.

[49]Το ακρωνύμιο "MVCC" του όρου "Multi-version Concurrency Control" αναφέρεται στην έννοια που θα μπορούσε να αποδοθεί στα ελληνικά ως "Έλεγχος Παραλληλισμού με Χρήση Πολλαπλών Εκδόσεων".

[50]Το ακρωνύμιο "ACID" αναφέρεται στους αγγλικούς όρους "Atomicity, Consistency, Isolation, Durability", οι οποίοι αποδίδονται στα ελληνικά ως "Ατομικότητα, Συνέπεια, Απομόνωση, Μονιμότητα".

Αποθήκευση με Διευθυνσιοδότηση-Βάσει-Περιεχομένου

Η αποθήκευση με διευθυνσιοδότηση-βάσει-περιεχομένου είναι ένας μηχανισμός αποθήκευσης πληροφορίας, με τρόπο ώστε η εκ των υστέρων προσπέλασή της να γίνεται λογικά βάσει του περιεχομένου της, και όχι της ακριβούς της θέσης [153]. Οι δύο παρακάτω παράγραφοι πηγάζουν από το [153], το οποίο περιγράφει την εν λόγω έννοια με ακρίβεια, αντιπαραβάλλοντάς την με παραδοσιακούς μηχανισμούς αποθήκευσης.

Στα πλαίσια της σύγκρισής τους με την αποθήκευση βάσει περιεχομένου, οι παραδοσιακές τοπικές ή δικτυακές μονάδες αποθήκευσης αναφέρονται στο εξής ως μονάδες "αποθήκευσης-βάσει-τοποθέτησης". Σε μια τέτοια μονάδα αποθήκευσης, κάθε στοιχείο δεδομένων αποθηκεύεται στο φυσικό μέσο, και η θέση του (λόγου χάριν το όνομα και η διαδρομή του) καταγράφονται σε έναν σχετικό κατάλογο για μετέπειτα χρήση τους. Με την έλευση ενός μελλοντικού αιτήματος για το στοιχείο αυτό, το αίτημα περιλαμβάνει μόνο την θέση των δεδομένων. Στη συνέχεια, η μονάδα αποθήκευσης μπορεί να χρησιμοποιήσει αυτή την πληροφορία για να εντοπίσει πού βρίσκονται τα δεδομένα στο φυσικό μέσο, και να τα προσπελάσει. Όταν νέα πληροφορία εγγράφεται σε μονάδες αποθήκευσης βάσει τοποθέτησης, αποθηκεύεται απλά σε κάποιο διαθέσιμο τμήμα του μέσου, ανεξάρτητα από το περιεχόμενό της. Η πληροφορία σε μία δεδομένη θέση μπορεί συνήθως να μεταβληθεί, ή και να επανεγγραφεί πλήρως, χωρίς να απαιτείται καμία ιδιαίτερη ενέργεια από την πλευρά της μονάδας αποθήκευσης [153].

Όταν η πληροφορία αποθηκεύεται σε σύστημα αποθήκευσης δεδομένων βάσει περιεχομένου, το σύστημα καταγράφει μία "διεύθυνση-βάσει-περιεχομένου", δηλαδή ένα μοναδικό αναγνωριστικό μονίμως συνδεδεμένο με το ίδιο το περιεχόμενο της εν λόγω πληροφορίας. Αυτή τη φορά, ένα αίτημα για προσπέλαση της πληροφορίας από το σύστημα πρέπει να περιλαμβάνει το αναγνωριστικό αυτό, βάσει του οποίου το σύστημα θα μπορέσει να προσδιορίσει τη φυσική θέση των δεδομένων, ώστε να τα προσπελάσει. Από τη στιγμή που τα αναγνωριστικά βασίζονται στο περιεχόμενο της πληροφορίας, οποιαδήποτε μεταβολή της, επιφέρει και αλλαγή στο ίδιο το αναγνωριστικό. Γι'αυτόν το λόγο, σχεδόν όλες οι περιπτώσεις συστημάτων αποθήκευσης βάσει περιεχομένου δεν επιτρέπουν την επεξεργασία των δεδομένων αφού έχουν αποθηκευτεί, ενώ και η διαγραφή τους συχνά εξαρτάται από την εκάστοτε πολιτική που ακολουθείται [153].

Εξ ορισμού της έννοιας της αποθήκευσης βάσει περιεχομένου, καθίσταται πασιφανές ότι συνταιριάζεται κατά πολύ φυσικό τρόπο με την έννοια της αμεταβλητότητας που εξετάσαμε νωρίτερα. Πράγματι, συστήματα αποθήκευσης βάσει περιεχομένου συνήθως στοχεύουν στην αποθήκευση δεδομένων που δεν αλλάζουν με την πάροδο του χρόνου. Σχεδιάζονται ώστε να καθιστούν την αναζήτηση και προσπέλαση για κάποιο δεδομένο περιεχόμενο πολύ γρήγορη, καθώς και την επιβεβαίωση ότι η πληροφορία που προσπελάζεται είναι πράγματι η ίδια που είχε νωρίτερα αποθηκευτεί – ειδάλλως το αναγνωριστικό θα διέφερε. Επιπροσθέτως, αφού τα δεδομένα αποθηκεύονται στο σύστημα βάσει του περιεχομένου τους, δεν μπορεί ποτέ να υπάρχει διπλότυπο, αφού δύο στοιχεία πληροφορίας ακριβώς ίδιου περιεχομένου που αποθηκεύονται στο σύστημα, θα έχουν αναγκαστικά και ακριβώς ίδιο αναγνωριστικό, το οποίο και θα δείχνει στην μοναδική θέση που είναι αποθηκευμένο το αντίστοιχο περιεχόμενο. Για δεδομένα που υφίστανται συχνές αλλαγές, η αποθήκευση βάσει περιεχομένου δεν είναι τόσο αποδοτική όσο η αποθήκευση βάσει τοποθέτησης. Σ'αυτές τις περιπτώσεις, θα ήταν αναγκαίο να επανυπολογίζονται διαρκώς τα αναγνωριστικά βάσει των νέων αλλαγμένων περιεχομένων, και οι πελάτες του συστήματος θα ήταν αναγκασμένοι να ενημερώνονται διαρκώς για τα νέα αυτά αναγνωριστικά. Για συστήματα τυχαίας προσπέλασης, ένα σύστημα αποθήκευσης βάσει περιεχομένου θα έπρεπε να μπορεί να χειρίζεται και την πιθανότητα δύο αποθηκευμένα στοιχεία πληροφορίας τα οποία αρχικώς είχαν ίδιο περιεχόμενο, σταδιακά να αποκλίνουν λόγω διαδοχικών μεταβολών τους, οπότε και θα απαιτούνταν η δημιουργία κάποιου επιπρόσθετου αντιγράφου για την απόκλιση αυτή, όποτε αυτή συμβαίνει.

Σήμερα, υπάρχουν πολλές περιπτώσεις στις οποίες χρειάζεται να αποθηκευτούν αμετάβλητα μεγάλα δυαδικά αντικείμενα, κάνοντας τη χρήση των συστημάτων αποθήκευσης βάσει περιεχομένου μία ενδεχόμενη λύση. Παραδείγματος χάριν, ποικίλες υπηρεσίες κοινωνικής δικτύωσης φιλοξενούν πολύ μεγάλο όγκο δεδομένων, είτε κειμένου είτε οπτικοακουστικού υλικού, τα οποία αποτελούν ιδανικούς "υποψηφίους" για αυτού του είδους τα συστήματα αποθήκευσης, αφού κατά κανόνα δημιουργούναι άπαξ, και προσπελάζονται πολλές φορές μόνο για ανάγνωση, μέχρι τελικά τη διαγραφή τους – το ίδιο το περιεχόμενό τους δεν μεταβάλλεται πρακτικά ποτέ. Σε τέτοιες περιπτώσεις είναι που η αποθήκευση βάσει περιεχομένου παρέχει και απαλοιφή διπλοτύπων, καταφέρνοντας με αυτόν τον τρόπο να μετριάσει την χρήση αποθηκευτικού χώρου, και επιτρέποντας στους κατάλληλους μηχανισμούς δημιουργίας αντιγράφων και εξι-

σορρόπησης φόρτου να διαχειριστούν οργανωμένα και αποδοτικά ενδεχόμενα σημεία συγκέντρωσης περιζήτητων δεδομένων, τα οποία συχνά παρουσιάζονται σε τέτοιου είδους κοινωνικά δίκτυα λόγω κοινωνικών τάσεων και ρευμάτων. Τα επιχειρήματα αυτά ισχύουν και για διάφορες υπηρεσίες ή πλατφόρμες αλληλογραφίας ή διαμοιρασμού αρχείων, οι οποίες είναι συχνά δημοφιλείς με τη μορφή SaaS[51]. Ένα άλλο παράδειγμα που η αποθήκευση βάσει περιεχομένου αποτελεί καλή λύση, είναι ο έλεγχος εκδόσεων: ουσιαστικά, το Git[110] είναι ένα σύστημα αποθήκευσης αρχείων βάσει περιεχομένου με διεπαφή συστήματος ελέγχου εκδόσεων. Οι καταστάσεις όλων των αρχείων αποθηκεύονται, και τους δίνονται μοναδικά αναγνωριστικά βάσει του περιεχομένου τους, έτσι ώστε να είναι δυνατόν εκ των υστέρων η αναφορά σε αυτά μέσω πολλών δέντρων αρχείων, παρά το γεγονός ότι αποθηκεύονται μόνο σε μία και μοναδική θέση.

Είναι προφανές ότι η αποθήκευση βάσει περιεχομένου ταιριάζει πολύ καλά στο μοντέλο των δεδομένων και της προσπέλασής τους που υιοθετείται από το σύστημά μας. Η έλλειψη υποστήριξης πράξεων μεταβολών είναι εν προκειμένω βολική, και του επιτρέπει να βασιστεί σ᾽αυτή την τεχνική. Συνεπώς, η σχεδίασή μας ενστερνίζεται την έννοια της αποθήκευσης βάσει περιεχομένου, και τη χρησιμοποιεί στην καρδιά της λειτουργικότητας του συστήματός μας. Το περιεχόμενο κάθε εισερχόμενου μεγάλου δυαδικού αντικειμένου εξετάζεται, και του ανατίθεται ένα μοναδικό αναγνωριστικό, το οποίο και χρησιμοποιείται ως το κλειδί στο ζεύγος κλειδιού-τιμής που δημιουργείται και αποθηκεύεται για το αντικείμενο αυτό. Αυτό το κλειδί (το αναγνωριστικό βάσει περιεχομένου) εμπεριέχεται στην απάντηση του συστήματος στον πελάτη, ούτως ώστε ο πελάτης να είναι σε θέση αργότερα να το χρησιμοποιήσει για να προσπελάσει ξανά το αντικείμενο όποτε χρειαστεί.

Κατακερματισμός

Μία βασική ερώτηση που εγείρεται φυσικώς και πλανάται ακόμα αναπάντητη πάνω από την προηγούμενη ενότητα, σχετικά με την αποθήκευση βάσει περιεχομένου, έχει να κάνει με τον τρόπο με τον οποίον τα μοναδικά αυτά αναγνωριστικά των δεδομένων δημιουργούνται και ανατίθενται στα εισερχόμενα μεγάλα δυαδικά αντικείμενα. Το πε-

[51] Ακρωνύμιο για τον αγγλικό όρο "Software-as-a-Service", δηλαδή "Λογισμικό-ως-Υπηρεσία".

ριεχόμενο κάθε τέτοιου αντικειμένου δίνεται ως είσοδο σε μία συνάρτηση κατακερματισμού[154]. Το παραγόμενο αποτέλεσμα της συνάρτησης κατακερματισμού, το οποίο στο εξής θα λέγεται "σύνοψη κατακερματισμού", χρησιμοποιείται ως τέτοιο αναγνωριστικό: το κλειδί του ζεύγους κλειδιού-τιμής που αποθηκεύεται. Οι συναρτήσεις κατακερματισμού αντιστοιχίζουν ακολουθίες ψηφιολέξεων μεταβλητού μεγέθους σε ακολουθίες ψηφιολέξεων σταθερού πλήθους, και συνεπώς όλες οι συνόψεις κατακερματισμού είναι ίδιου μήκους.

Δεν είναι όλες οι συναρτήσεις κατακερματισμού ισοδύναμες, και, όπως είναι αναμενόμενο, η επιλογή της καταλληλότερης είναι μία σημαντική απόφαση. Η αποθήκευση με διευθυνσιοδότηση βάσει περιεχομένου απαιτεί μία συνάρτηση κατακερματισμού με "αντοχή σε συγκρούσεις". Ειδάλλως, διαφορετικά στοιχεία αποθηκευμένης πληροφορίας ενδεχομένως να συγκρούονται, με την έννοια ότι μπορεί να τους ανατίθεται το ίδιο αναγνωριστικό βάσει του περιεχομένου τους. Η αντοχή στις συγκρούσεις δεν σημαίνει ότι συγκρούσεις δεν γίνονται ποτέ – η αρχή του περιστερώνα εγγυάται ότι αφού ο πληθικός αριθμός των τιμών εισόδου είναι μεγαλύτερος από αυτόν των τιμών εξόδου, αναπόφευκτα κάποιες είσοδοι θα αντιστοιχίζονται σε κοινή έξοδο. Σημαίνει απλώς ότι η πιθανότητα να γίνει τέτοια σύγκρουση είναι πολύ μικρή.

Για το λόγο αυτό, τα περισσότερα συστήματα αποθήκευσης με διευθυνσιοδότηση βάσει περιεχομένου εκθέτουν συνόψεις κατακερματισμού που παράγονται με κρυπτογραφικές συναρτήσεις κατακερματισμού[155] από τα δεδομένα στα οποία αναφέρονται. Αυτές αποτελούν μία ειδική κατηγορία συναρτήσεων κατακερματισμού, με κάποιες επιπρόσθετες ιδιότητες. Κάποιες από αυτές τις ιδιότητες είναι ότι οι έξοδοι των κρυπτογραφικών συναρτήσεων κατακερματισμού είναι ντετερμινιστικές, αδύνατον να αντιστραφούν, γρήγορα υπολογίσιμες, και διέπονται από το "φαινόμενο της χιονοστιβάδας"[52] [155]. [155]. Ακόμα, είναι "δύσκολα αντριστρέψιμες"[53] και έχουν αντοχή σε συγκρούσεις [156]. Παραδείγματα συστημάτων αποθήκευσης με διευθυνσιοδότηση βάσει περιεχομένου που χρησιμοποιούν κρυπτογραφικές συναρτήσεις κατακερματισμού είναι τα Git[110], Venti[157] του Plan9, και CASPER[158].

[52] Μικρές τροποποιήσεις στην είσοδο της συνάρτησης κατακερματισμού, όπως η ανάρριψη ενός μόνο δυφίου, προκαλεί σημαντική αλλαγή στην έξοδό της. Με άλλα λόγια, γενικώς, παρόμοιες – αλλά όχι εντελώς ίδιες – είσοδοι παράγουν εντελώς διαφορετικές και ασυνάρτητες μεταξύ τους εξόδους.

[53] Δεδομένης μιας σύνοψης κατακερματισμού N διφύων, παραγμένη από κρυπτογραφική συνάρτηση κατακερματισμού, θεωρητικά χρειάζεται πλήθος υπολογισμών της τάξεως του 2^N για να βρεθεί είσοδος που έχει την ίδια σύνοψη κατακερματισμού.

Μία σημαντική ιδιότητα των συναρτήσεων κατακερματισμού, με πρακτική και θεωρητική αξία για την περίπτωσή μας – αφού πρόκειται να μας βοηθήσει στη μαθηματική μοντελοποίηση και στην συλλογιστική του προβλήματος της επιλογής συνάρτησης κατακερματισμού – είναι η ομοιομορφία. Μία ομοιόμορφη συνάρτηση κατακερματισμού αντιστοιχίζει τις αναμενόμενες εισόδους της όσο πιο "δίκαια" και ομοιόμορφα γίνεται στο εύρος των εξόδων. Δηλαδή, κάθε σύνοψη κατακερματισμού στο εύρος εξόδων παράγεται με περίπου την ίδια πιθανότητα όπως οι υπόλοιπες [154], ώστε να διαμορφώνει μία διακριτή ομοιόμορφη κατανομή[54]. Αυτή η ιδιότητα είναι προφανώς χρήσιμη σε ό,τι αφορά την αντοχή σε συγκρούσεις, αφού ένα ανομοιόμορφα κατανεμημένο εύρος εξόδων συνεπάγεται υποδιαστήματα υψηλής συγκέντρωσης επί του συνολικού εύρους εξόδων, όπου η πιθανότητα δύο οποιωνδήποτε συνόψεων κατακερματισμού να συγκρουστούν έχει προφανώς μεγαλύτερη τιμή.

Το μέγεθος των παραγόμενων συνόψεων κατακερματισμού συνιστά έναν άλλον καθοριστικό παράγοντα σχετικό με την αντοχή της αντίστοιχης συνάρτησης κατακερματισμού σε συγκρούσεις. Αποδεικνύεται ότι το πρόβλημα των συγκρούσεων αποτελεί μία γενίκευση του προβλήματος των Γενεθλίων[160, 161], το οποίο μπορεί να τεθεί ως εξής:

> *Δεδομένων k τυχαίων ακέραιων αριθμών που προέρχονται από μία διακριτή ομοιόμορφη κατανομή στο εύρος $[0, n-1]$, ποια είναι η πιθανότητα $P(k;n)$ τουλάχιστον δύο εξ αυτών να είναι οι ίδιοι;*

Στην περίπτωσή μας, το συνολικό πλήθος μοναδικά ανατεθημένων αναγνωριστικών βάσει περιεχομένου αναπαρίσταται με k, και, για μία συνάρτηση κατακερματισμού που παράγει συνόψεις των N δυφίων, θα ίσχυε $n = 2^N$. Το $P(k;n)$ εκφράζει την πιθανότητα να συμβεί μία σύγκρουση μεταξύ δύο οποιωνδήποτε αναγνωριστικών βάσει περιεχομένου.

Μπορεί να αποδειχθεί ότι

$$P(k;n) = 1 - \prod_{i=1}^{k-1}(1 - \frac{i}{n}) \tag{1}$$

[54]Στη Θεωρία Πιθανοτήτων και τη Στατιστική, η διακριτή ομοιόμορφη κατανομή είναι μία συμμετρική κατανομή πιθανότητας όπου όλες οι τιμές εντός ενός πεπερασμένου πλήθους τιμών είναι εξίσου πιθανό να παρατηρηθούν. Καθεμία από τις n τιμές έχει ίση πιθανότητα $\frac{1}{n}$. Ένας εναλλακτικός τρόπος να αναφερθούμε στην "διακριτή ομοιόμορφη κατανομή" θα ήταν "ένα γνωστό, πεπερασμεο πλήθος αποτελεσμάτων, εξίσου πιθανό να συμβούν" [159].

όπου $k \leq n$, με

$$P(k;n) \approx 1 - e^{-\frac{k(k-1)}{2n}} \tag{2}$$

μία ικανοποιητική προσέγγιση της ίδιας τιμής.

Εναλλακτικά, για απλότητα, μπορούμε να σκεφτούμε την πιθανότητα $P(k;n)$ περιορισμένη μεταξύ του πλήθους όλων των ζευγών τυχαίων ακέραιων αριθμών, πολλαπλασιασμένο με την πιθανότητα σύγκρουσης ενός τέτοιου ζεύγους [157], δηλαδή:

$$P(k;n) \leq \frac{k(k-1)}{2} \times \frac{1}{n} \tag{3}$$

Ξεκινούμε εξετάζοντας το εύρος των τιμών του k στην περίπτωσή μας. Θεωρώντας συνολική χωρητικότητα 1 PiB μοναδικής πληροφορίας (όπως έχουμε αποσαφηνίσει ήδη, εννοώντας εξαιρουμένων των αντιγράφων), και δεδομένου ότι καθένα εξ αυτών είναι κατά μέσο όρο 4 MiB, το σύστημά μας θα πρέπει να είναι σε θέση να φιλοξενήσει τουλάχιστον $2^{28} \approx 2.7 \times 10^8$ αντικείμενα. Φυσικά, αφού κάποια από τα αντικείμενα μπορεί να είναι μικρότερα, το συνολικό τους πλήθος μπορεί να είναι αρκετά μεγαλύτερο. Παραδείγματος χάριν, για αντικείμενα της τάξης των 4 KiB το καθένα κατά μέσο όρο, το σύστημα θα πρέπει να είναι σε θέση να φιλοξενήσει περίπου $2^{38} \approx 2.75 \times 10^{11}$ τέτοια αντικείμενα.

Στη συνέχεια, εξετάζουμε τις τιμές του n. Όπως έχει προαναφερθεί, $n = 2^N$ για οποιαδήποτε συνάρτηση κατακερματισμού που παράγει συνόψεις κατακερματισμού των N δυφίων. Κατ'αρχάς, ενστικτωδώς απορρίπτουμε όλες τις συναρτήσεις κατακερματισμού 32 δυφίων, αφού το $n = 2^{32}$ είναι πολύ απλά ανεπαρκές, ακόμα για τις ελάχιστες αναμενόμενες τιμές του k (δηλαδή, περίπου $\approx 2^{28}$ αντικείμενα). Εκτός αυτών, υπάρχουν πολλές ομοιόμορφες συναρτήσεις κατακερματισμού με συνόψεις 64 δυφίων – ή και περισσότερων – από τις οποίες μπορούμε να επιλέξουμε. Στο σημείο αυτό οφείλουμε να επισημάνουμε ότι καμία από τις διαθέσιμες επιλογές δεν είναι πραγματικά απαλλαγμένη μειονεκτημάτων. Αφού συνόψεις κατακερματισμού N δυφίων χρησιμοποιούνται ως κλειδιά στα ζεύγη κλειδιού-τιμής, ούτως ή άλλως εν τέλει και αυτές αποθηκεύονται στο σύστημα. Σε μία ακραία περίπτωση, όπου έχουμε $k \approx 2^{38}$ και επιλέγεται μία ομοιόμορφη συνάρτηση κατακερματισμού, με αντοχή σε συγκρούσεις, και με συνόψεις 512 δυφίων, ο αποθηκευτικός χώρος που απαιτείται μόνο για τα κλειδιά των ζευγών φτάνει τα 16 TiB. Αυτό είναι περίπου το 1.56% επιπλέον της συνολικής

χωρητικότητας του συστήματος, το οποίο είναι υπερβολικά υψηλό ποσοστό, ιδίως αναλογιζόμενοι πως τα κλειδιά καθαυτά δεν φέρουν ουδεμία σημασιολογική ή άλλου είδους χρήσιμη για τους χρήστες πληροφορία – ο μοναδικός τους σκοπός είναι να διευκολύνουν την υλοποίηση της αποθήκευσης με διευθυνσιοδότηση βάσει περιεχομένου στο σύστημά μας. Συνεπώς, ενστικτωδώς, απορρίπτουμε και όλες τις συναρτήσεις κατακερματισμού 512 δυφίων, αφού το $n = 2^{512}$ φαίνεται υπερβολικά υψηλό ακόμα και για τις μεγαλύτερες αναμενόμενες τιμές του k, και αφετέρου το κόστος για την τόσο σημαντική μείωση της πιθανότητας συγκρούσεων – δηλαδή το κόστος της αποθήκευσης των κλειδιών σε χωρητικότητα – κρίνεται ανισόρροπα υψηλό για τις περισσότερες περιπτώσεις χρήσης του συστήματος.

Βασιζόμενοι σ'αυτές τις παρατηρήσεις και χρησιμοποιώντας μία εκ των (1), (2), (3), μπορούμε τώρα να υπολογίσουμε τις τιμές της πιθανότητας σύγκρουσης για διαφορετικές τιμές των n και k. Αυτοί οι υπολογισμοί περιλαμβάνουν τιμές του N στο εύρος $[64, 384]$ που χρησιμοποιούνται ευρέως από πληθώρα συναρτήσεων κατακερματισμού, και δύο τιμές για το k. Οι τιμές καταρτίζουν τον παρακάτω Πίνακα.

N	**n**	$\mathbf{P(k; n)}$	
		$\mathbf{k = 2^{28} \approx 2.68 \times 10^8}$	$\mathbf{k = 2^{38} \approx 2.75 \times 10^{11}}$
64	$2^{64} \approx 1.8 \times 10^{19}$	2×10^{-3}	1
128	$2^{128} \approx 3.4 \times 10^{38}$	1.06×10^{-22}	1.11×10^{-16}
160	$2^{160} \approx 1.5 \times 10^{48}$	2.47×10^{-32}	2.58×10^{-26}
224	$2^{224} \approx 2.7 \times 10^{67}$	1.34×10^{-51}	1.4×10^{-45}
256	$2^{256} \approx 1.2 \times 10^{77}$	3.11×10^{-61}	3.26×10^{-55}
384	$2^{384} \approx 3.9 \times 10^{115}$	9.14×10^{-100}	9.59×10^{-94}

Πίνακας 1: *Οι τιμές της πιθανότητας σύγκρουσης μεταξύ δύο οποιωνδήποτε συνόψεων κατακερματισμού, $P(k; n)$, για ευρέως χρησιμοποιούμενες τιμές του n και δύο τιμές του k.*

Αντίστροφα, από την (2) είναι δυνατόν να εξάγουμε έναν τύπο για την προσέγγιση της τιμής του k, του πλήθους των αντικειμένων που αποθηκεύονται στο σύστημα, δεδομένης μίας ομοιόμορφης συνάρτησης κατακερματισμού με συνόψεις των N δυφίων, και την επιθυμητή τιμή της πιθανότητας σύγκρουσης μεταξύ δύο οποιωνδήποτε συνόψεων κατακερματισμού στο εύρος $[0, n - 1]$, όπου $k \leq n$:

$$k(P; n) \approx \sqrt{2n \ln\left(\frac{1}{1 - P}\right)} \tag{4}$$

όπου $n = 2^N$.

Εν συνεχεία, χρησιμοποιώντας την (4) και βασιζόμενοι στις παραπάνω τιμές, είναι εύκολο να υπολογίσουμε τις μέγιστες τιμές του k, για τις ίδιες τιμές των N και n, δεδομένου ότι η επιθυμητή τιμή της πιθανότητας να συμβεί μια τέτοια σύγκρουση είναι $P = 10^{-18}$. Τα αποτελέσματα καταρτίζουν τον παρακάτω Πίνακα.

N	**n**	**k**
64	$2^{64} \approx 1.8 \times 10^{19}$	12
128	$2^{128} \approx 3.4 \times 10^{38}$	$2.6 \times 10^{10} \approx 1.51 \times 2^{34}$
160	$2^{160} \approx 1.5 \times 10^{48}$	$1.7 \times 10^{15} \approx 1.5 \times 2^{50}$
224	$2^{224} \approx 2.7 \times 10^{67}$	$7.34 \times 10^{24} \approx 1.51 \times 2^{82}$
256	$2^{256} \approx 1.2 \times 10^{77}$	$4.81 \times 10^{29} \approx 1.51 \times 2^{98}$
384	$2^{384} \approx 3.9 \times 10^{115}$	$8.87 \times 10^{48} \approx 1.51 \times 2^{162}$

Πίνακας 2: *Ασφαλείς για χρήση τιμές του $k(P; n)$, δηλαδή του μέγιστου πλήθους αποθηκευμένων αντικειμένων στο σύστημα, για ευρέως χρησιμοποιούμενες τιμές του n, δεδομένου ότι η επιθυμητή τιμή της πιθανότητας σύγκρουσης μεταξύ δύο οποιωνδήποτε συνόψεων κατακερματισμού είναι $P = 10^{-18}$.*

Από την πρώτη κιόλας βιαστική ματιά των δύο πινάκων, γίνεται σαφές ότι οι συναρτήσεις κατακερματισμού 64 δυφίων δεν είναι έμπιστες για την αποφυγή συγκρούσεων όταν το πλήθος των αποθηκευμένων αντικειμένων στο σύστημά μας είναι μέγιστο. Ακόμα και αν μειώσουμε την επιθυμητή τιμή της πιθανότητας σύγκρουσης δύο οποιωνδήποτε συνόψεων κατακερματισμού στην σχετικά χαμηλή τιμή $P = 10^{-10}$, θα ήταν και πάλι παρακινδυνευμένο να αποθηκεύσουμε πάνω από περίπου $60000 < 2^{16}$ αντικείμενα, το οποίο είναι γενικά ένας αρκετά μικρός αριθμός ως επιτρεπτό πλήθος αντικειμένων.

Από την άλλη, συναρτήσεις κατακερματισμού 384 δυφίων, αλλά και 224 και 256 δυφίων, φαίνεται ίσως να ξεπερνούν τις απαιτήσεις των στόχων μας. Παρότι η μείωση της πιθανότητας συγκρούσεων όσο το δυνατόν περισσότερο είναι σίγουρα επιθυμητή, οφείλουμε να συνυπολογίζουμε πάντα και τις αρνητικές επιπτώσεις μιας τέτοιας επιλογής, οι οποίες αναφέρθηκαν παραπάνω. Συναρτήσεις κατακερματισμού των 128 και 160 δυφίων φαίνονται να ακροβατούν στα αποδεκτά μας όρια: παρότι φαίνονται αρκετά ασφαλείς για χρήση με $k \approx 2.68 \times 10^8$ αποθηκευμένα αντικείμενα, για τιμές του k της τάξης του $\sim 2^{48} \approx 2.8 \times 10^{14}$, μια τέτοια επιλογή θα μπορούσε να επιφέρει κινδύνους.

Πριν επιλέξουμε μία συνάρτηση κατακερματισμού, οφείλουμε να αναλογιστούμε το

ζήτημα της επιλογής και από τη σκοπιά της ασφάλειας. Μία δύσκολα αντιστρέψιμη συνάρτηση κατακερματισμου με αντοχή σε συγκρούσεις, όπως λόγου χάριν οι κρυπτογραφικές, μπορεί να χρησιμοποιηθεί για να παράσχει κάποια εχέγγυα αναφορικά με την ασφάλεια τόσο του συστήματος, όσο και των πελατών του. Από την πλευρά του συστήματος, μπορεί να εξουδετερώσει σε σημαντικό βαθμό κακόβουλους πελάτες που επιδιώκουν να δημιουργήσουν αντικείμενα με τέτοιο τρόπο ώστε να προσβάλλεται η προϋπόθεση ότι κάθε αντικείμενο αντιστοιχίζεται σε ένα μοναδικό αναγνωριστικόκλειδί. Από την πλευρά ενός πελάτη, επιτρέπει ισχυρότερους ελέγχους ακεραιότητας των αποθηκευμένων δεδομένων, αποτρέποντας ενδεχόμενη κακόβουλη συμπεριφορά του συστήματος, το οποίο θα μπορούσε ενδεχομένως να απαντά σε αιτήματα ανάγνωσης με "λάθος" – πιθανώς κατασκευασμένα – δεδομένα.

Από τα παραπάνω, συμπεραίνουμε ότι η επιλογή της κατάλληλης συνάρτησης κατακερματισμού εξαρτάται σε μεγάλο βαθμό από τις απαιτήσεις που τίθενται από την εκάστοτε περίπτωση χρήσης του συστήματος, η οποία πιθανώς με τη σειρά της να καθορίζεται από μία υψηλότερου επιπέδου λογική ή αρχιτεκτονική κάθε φορά. Το κατανεμημένο σύστημα αποθήκευσης αντικειμένων της παρούσας εργασίας είναι σχεδιασμένο και υλοποιημένο με τέτοιο τρόπο, ώστε να λειτουργεί ανεξαρτήτως της συνάρτησης κατακερματισμού που χρησιμοποιείται. Παρότι η αντοχή σε συγκρούσεις, και άρα και η έννοια της αποθήκευσης με διευθυνσιοδότηση βάσει περιεχομένου, εξαρτάται σε πολύ μεγάλο βαθμό από την ομοιομορφία της συνάρτησης κατακερματισμού και το μέγεθος των συνόψεων που αυτή παράγει, κάθε άλλη πτυχή του συστήματος, αναφορικά με διάφορες άλλες συνιστώσες της αρχιτεκτονικής του, μένει ανεπηρέαστη από την επιλογή αυτή. Αυτός ο τρόπος σχεδίασης επιτρέπει εύκολη αναβάθμιση του συστήματος στο μέλλον, και παρέχει ευελιξία για χρήσεις σε μεγαλύτερη ποικιλία περιπτώσεων.

Τεμαχισμός

Μέχρι στιγμής έχουμε συζητήσει για την αμεταβλητότητα, την αποθήκευση με διευθυνσιοδότηση βάσει περιεχομένου, και τον κατακερματισμό, καθώς και το πώς όλα αυτά συνταιριάζονται στη σχεδίαση του συστήματος αποθήκευσης της παρούσας εργασίας. Σημειώνεται, μολαταύτα, ότι κανένα από τα παραπάνω δεν έχει να κάνει συ-

γκεκριμένα με τη Θεωρία Κατανεμημένων Συστημάτων. Τώρα, θα παρουσιάσουμε μία ακόμα σπουδαία έννοια, που έχει σημαντικό ρόλο στον πυρήνα του συστήματός μας, και η οποία θεωρείται ευρέως ως βασικός πυλώνας της οριζόντιας κλιμακωσιμότητας και της ελαστικότητας των σύγχρονων κατανεμημένων συστημάτων αποθήκευσης.

Με τον όρο "διαμέριση βάσης δεδομένων"[55] αναφερόμαστε στη διαδικασία κατά την οποία μία λογική βάση δεδομένων διαμερίζεται σε ανεξάρτητα τμήματα για λόγους επιδόσεων, διαθεσιμότητας, εξισορρόπησης φόρτου ή απλά διαχειρισιμότητας [177]. Παραδοσιακά, η διαμέριση μπορεί να είναι είτε κατακόρυφη είτε οριζόντια, αλλά εμείς δεν πρόκειται να ασχοληθούμε με το πρώτο είδος καθόλου. Στην "οριζόντια διαμέριση", τα δεδομένα χωρίζονται σε πολλαπλές διαφορετικές μονάδες, συχνά βάσει κάποιου κλειδιού – συνήθως του κλειδιού της κάθε εγγραφής ή αντικειμένου της βάσης δεδομένων. Αυτή η μέθοδος έχει πολλά πλεονεκτήματα, κυρίως από την σκοπιά της αποδοτικότητας: τα μεγέθη των ευρετηρίων είναι μειωμένα, και τα ερωτήματα μπορούν να κατευθύνονται στην κατάλληλη κάθε φορά διαμέριση, έχοντας ως αποτέλεσμα την βελτίωση της απόδοσης των αναζητήσεων.

Με τον όρο "τεμαχισμό"[56] αναφερόμαστε στην ειδική περίπτωση οριζόντιας διαμέρισης, όπου οι διαμερίσεις, οι οποίες πλέον ονομάζονται και "τεμάχια", μπορούν να είναι κατανεμημένες σε πολλαπλούς φυσικούς κόμβους, σε διαφορετικές πιθανώς τοποθεσίες ο καθένας. Αυτοί οι κόμβοι πρέπει να μπορούν να λειτουργούν κατάλληλα ακόμα και όταν είναι απομονωμένοι, γεγονός το οποίο απαιτεί ιδιαίτερη προσοχή σε σχέση με την απλή οριζόντια διαμέριση, ειδάλλως τα επιθυμητά κέρδη όσον αφορά την αποδοτικότητα μπορεί να χαθούν αν, παραδείγματος χάριν, οι ερωτήσεις προς το σύστημα για προσπέλαση δεδομένων ενδεχομένως απαιτούν την αποστολή ερωτημάτων σε πολλαπλούς φυσικούς κόμβους αντί για έναν [178]. Αυτός είναι και ο λόγος που ο τεμαχισμός είναι συχνά συνδεδεμένος με "αρχιτεκτονικές μηδενικού διαμοιρασμού"[57][58] – από τη στιγμή του τεμαχισμού, κάθε τεμάχιο θα πρέπει να λειτουργεί ανε-

[55]Με τον όρο "διαμέριση βάσης δεδομένων" αναφερόμαστε στην έννοια που εκφράζει ο αγγλικός όρος "database partitioning".

[56]Με τον ελληνικό όρο "τεμαχισμός" αναφερόμαστε στην έννοια που εκφράζεται με τον αγγλικό όρο "sharding".

[57]Με τον ελληνικό όρο "αρχιτεκτονική μηδενικού διαμοιρασμού" προσπαθούμε να εκφράσουμε την έννοια του αγγλικού όρου "shared-nothing architecture".

[58]Μία "αρχιτεκτονική μηδενικού διαμοιρασμού είναι μία αρχιτεκτονική για περιβάλλοντα κατανεμεμένων συστημάτων όπου κάθε κόμβος είναι ανεξάρτητος και αυτόχθων – οι κόμβοι δεν μοιράζονται μεταξύ τους ούτε μνήμη, ούτε αποθηκευτικά μέσα, και δεν υπάρχει κανένα μοναδικό σημείο συμφόρησης σε όλη την έκταση του συστήματος [179].

ξάρτητα από τα υπόλοιπα, και πιθανώς διαχωρισμένο απ᾽αυτά. Αυτό προσθέτει την οριζόντια κλιμακωσιμότητα στη λίστα των πλεονεκτημάτων του τεμαχισμού, αφού διαφορετικοί κόμβοι μπορούν να ανατεθούν για την αποθήκευση και απάντηση ερωτημάτων σχετικά με ένα συγκεκριμένο τεμάχιο – ή ένα σύνολο τέτοιων – και το πλήθος των κόμβων μπορούν να προσαρμόζονται στις εκάστοτε ανάγκες για κλιμακωσιμότητα. Επιπροσθέτως, η προϋποτιθέμενη απομόνωση στην αρχιτεκτονική συνήθως υποβοηθά την συλλογιστική περί τεχνικών αυτόματης δημιουργίας αντιγράφων, όποτε απαιτείται κάτι τέτοιο. Οι διαμερίσεις μπορεί να είναι είτε ξένα μεταξύ τους σύνολα, δηλαδή κάθε στοιχείο αποθηκευμένων στο σύστημα δεδομένων να εμφανίζεται σε ένα μόνο τεμάχιο και και ο κόμβος στον οποίο έχει ανατεθεί να είναι η μοναδική πηγή γι᾽αυτό, είτε να εφαρμόζεται κάποιο πιο σύνθετο σχήμα δημιουργίας αντιγράφων, στο οποίο πολλαπλά αντίγραφα της ίδιας εγγραφής ή αντικειμένου μπορεί να είναι παρόντα σε πολλαπλά διαφορετικά τεμάχια.

Το κατανεμημένο σύστημα αποθήκευσης της εργασίας μας είναι σχεδιασμένο να εκτελείται πάνω από ευρέως εμπορευματοποιημένους σκληρούς δίσκους και δίσκους στερεάς κατάστασης. Σήμερα, ενδεικτικές τιμές για τη χωρητικότητα τέτοιων δίσκων είναι περί τα 2 TiB έκαστος, μπορώντας έτσι να παράσχουν περίπου 16 TiB ανά κόμβο κέντρου δεδομένων. Είναι προφανές ότι, προκειμένου το σύστημά μας να μπορεί να φιλοξενήσει 1 PiB δεδομένων, τα δεδομένα πρέπει να είναι τεμαχισμένα και διαμοιρασμένα σε πολλούς κόμβους. Για παράδειγμα, ακόμα κι αν δεν δημιουργούνται καθόλου αντίγραφα των αποθηκευμένων δεδομένων για κάθε αντικείμενο του συστήματος, για να αποθηκευτούν αντικείμενα συνολικής χωρητικότητας 1 PiB σε κόμβους που παρέχουν 16 TiB έκαστος, απαιτούνται τουλάχιστον 64 τέτοιοι κόμβοι. Συνεπώς, το ελάχιστο πλήθος τεμαχίων που θα έπρεπε να δημιουργηθούν, χωρίς να συνυπολογιστεί οποιαδήποτε τεχνική δημιουργίας αντιγράφων προς το παρόν, είναι 64. Συνυπολογίζοντας έναν παράγοντα αντιγραφής k, το σύστημά μας θα έπρεπε να εκτελεστεί σε $64 \cdot k$ κόμβους του κέντρου δεδομένων. Ο ακριβής τρόπος τοποθέτησης των τεμαχίων στους κόμβους αυτούς, δε, απαιτεί ιδιαίτερη προσοχή, αφού αντίγραφα του ίδιου αντικειμένου που τοποθετούνται στον ίδιο κόμβο δεν συμβάλλουν ουσιαστικά στην ανοχή σφαλμάτων δίσκων ή κόμβων.

Ένα κρίσιμο ζήτημα σχετικά με τον τεμαχισμό είναι το πώς εξισορροπείται κατάλληλα ο φόρτος του συστήματος μεταξύ των κόμβων. Με άλλα λόγια, σε ποιον κόμβο πρέ-

πει να αποθηκευτεί το κάθε αντικείμενο ώστε όλοι οι κόμβοι να είναι ανεξάρτητοι μεταξύ τους και όλα τα αντικείμενα να είναι κατανεμημένα με ισορροπία μεταξύ των κόμβων αυτών; Και πώς ορίζεται αυτή η "ισορροπία" σε κάθε περίπτωση; Πρακτικά, η χωρητικότητα του κάθε κόμβου του κέντρου δεδομένων μπορεί να είναι διαφορετική, ιδίως σε ιδιωτικές υποδομές και περιβάλλοντα. Ιδανικά, τα αντικείμενα πρέπει να κατανέμονται αναλογικά ως προς τη χωρητικότητα κάθε κόμβου, και κατά συνέπεια το μέγεθος του τεμαχίου που ανατίθεται σε κάθε κόμβο πρέπει να αντιστοιχεί στη χωρητικότητα του κόμβου αυτού. Ενώ αυτό είναι πρακτικά εφικτό, ελοχεύουν κίνδυνοι που μπορεί να καταστήσουν τη διαδικασία σχετικά σύνθετη. Για το λόγο αυτό, στο πλαίσιο της παρούσας εργασίας, όπου έχουμε αναπτύξει δύο συστήματα με, αντιστοίχως, δύο διαφορετικούς αλγορίθμους τεμαχισμού, έχουμε υποθέσει ότι η χωρητικότητα όλων των κόμβων είναι ίση[59].

Ένα άλλο σημαντικό ζήτημα που σχετίζεται με τις μεθόδους τεμαχισμού, είναι η συμπεριφορά τους όταν οι κόμβοι πρέπει να προστεθούν ή να αφαιρεθούν από τη συστοιχία. "Επανατεμαχισμός"[60] είναι η διαδικασία της εκ νέου διαμέρισης μίας ήδη διαμερισμένης λογικής βάσης δεδομένων σε νέα τεμάχια. Μπορεί να υπάρχουν διάφορες αιτίες για τις οποίες αναφύεται η ανάγκη για επανατεμαχισμό. Παραδείγματος χάριν, τι συμβαίνει όταν το πλήθος των τεμαχίων στα οποία διαμερίστηκε το σύστημα αποθήκευσης είναι ίσο με το πλήθος των κόμβων που χρησιμοποιούνται για την εκτέλεση του συστήματος, αλλά η συνολική χωρητικότητα του συστήματος πρέπει οπωσδήποτε να αυξηθεί περαιτέρω, εκτελώντας, λοιπόν, το σύστημα σε ακόμα περισσότερους κόμβους; Υπάρχει προφανώς η ανάγκη περαιτέρω διαμέρισης των δεδομένων σε τεμάχια, και συνεπώς, με κάποιον τρόπο, οι ακριβείς κανόνες της διαδικασίας τεμαχισμού του συστήματος τροποποιούνται από εκείνη τη στιγμή και μετά. Πώς επηρεάζει αυτή η τροποποίηση τα δεδομένα τα οποία είναι ήδη παρόντα στο τρέχον πλήθος τεμαχίων; Καθώς τα τεμάχια διανέμονται στους φυσικούς κόμβους, τα δεδομένα τους είναι τρόπον τινά "δεμένα" στους φυσικούς κόμβους όπου βρίσκονταν πρωτύτερα – η μετακίνησή τους μπορεί να είναι δαπανηρή σε ό,τι αφορά τους δικτυακούς πόρους

[59]Παρ'ολ'αυτά, μία εκ των δύο τεχνικών τεμαχισμού είναι σχεδιασμένη και υλοποιημένη με τέτοιο τρόπο, ώστε να περιορίζει το ζήτημα αυτό σε μία από τις βιβλιοθήκες που το απαρτίζουν. Επεκτείνοντας αυτή τη βιβλιοθήκη, σε συνδυασμό με το κατάλληλο σύνολο ρυθμίσεων του Kubernetes, θα επέτρεπε ακριβή, "ευφυή" χειρισμό της διαδικασίας του τεμαχισμού με τρόπο αναλογικό ως προς τη χωρητικότητα κάθε κόμβου.

[60]Με τον ελληνικό όρο "επανατεμαχισμός" αναφερόμαστε στην έννοια που εκφράζει ο αγγλικός όρος "resharding".

και τις επιδόσεις γενικότερα, και φυσικά καταστροφική αν δεν πραγματοποιηθεί εξαιρετικά προσεκτικά. Τι ποσοστό των υπαρχόντων δεδομένων μετακινούνται στα νέα τεμάχια που διαμορφώνονται; Κατά τη διάρκεια της μετακίνησής τους, τα δεδομένα ρέουν μόνο προς τους νεοεισαχθέντες κόμβους του συστήματος, ή και μεταξύ των προϋπαρχόντων; Οι αλγόριθμοι αυτοματοποιημένης δημιουργίας και διανομής των αντιγράφων επηρεάζονται από τον επανατεμαχισμό, και αν ναι, τότε με ποιον τρόπο; Αυτά είναι μόνο μερικά από τα κεντρικά, υψηλού επιπέδου ερωτήματα που πρέπει να απαντώνται από έναν αλγόριθμο τεμαχισμού σχετικά με τον επανατεμαχισμό. Δεν υπάρχει καμία μοναδική σωστή απάντηση για κανένα από αυτά, και διαφορετικές πολιτικές μπορεί να ταιριάζουν καλύτερα σε διαφορετικές περιπτώσεις χρήσης.

Μεταξύ ποικίλων προσεγγίσεων για την επίτευξη του τεμαχισμού, επιλέγουμε εκείνες που χρησιμοποιούν τον κατακερματισμό ως βάση τους. Ο κατακερματισμός χρησιμοποιείται ήδη από το σύστημα για την υλοποίηση και ενσωμάτωση της έννοιας της αποθήκευσης με διευθυνσιοδότηση βάσει περιεχομένου, και συνεπώς είναι ήδη στον πυρήνα της σχεδίασής μας. Συνεπώς, η χρήση του για έναν ακόμα σκοπό δεν προκαλεί κανενός είδους επιβάρυνση. Μέσω μίας ομοιόμορφης συνάρτησης κατακερματισμού N δυφίων, τα κλειδιά των αντικειμένων – συνόψεις κατακερματισμού – είναι ομοιόμορφα κατανεμημένες τιμές στο εύρος $[0, 2^N - 1]$, το "εύρος κλειδιών" μας. Ο τεμαχισμός μπορεί να εξεταστεί απλώς ως το προβλημα της εύρεσης κατάλληλης διαμέρισης του εύρους κλειδιών. Η διαμέριση καθαυτή, καθώς και η διαδικασία ανάθεσης κάθε προκύπτοντος τμήματος σε φυσικό κόμβο, ποικίλει μεταξύ των δύο διαφορετικών τεχνικών τεμαχισμού που χρησιμοποιούνται στα δύο συστήματα της εργασίας. Το ίδιο και ικανότητα και η συμπεριφορά τους σχετικά με τον επανατεμαχισμό. Όμως, λεπτομέρειες σχετικά με καθεμία από τις τεχνικές αυτές, μαζί με τα πλεονεκτήματα και τα μειονεκτήματά τους, πρόκειται να συζητηθούν σύντομα, με την παρουσίαση του κάθε συστήματος ξεχωριστά.

Γενική Άποψη της Αρχιτεκτονικής

Προτού καταπιαστούμε με τις λεπτομέρειες της σχεδίασης των δύο συστημάτων, θεωρούμε εποικοδομητικό να εξηγήσουμε τις ομοιότητες και τα κοινά χαρακτηριστικά της αρχιτεκτονικής τους. Έως τώρα έχουμε εξετάσει ποικίλες έννοιες και τεχνικές που,

με τον έναν ή με τον άλλον τρόπο, χρησιμοποιούνται από διάφορα κατανεμημένα συστήματα αποθήκευσης. Στο σημείο αυτό, ήρθε η ώρα να τα βάλουμε σε μία σειρά, και να διαπιστώσουμε με ποιον τρόπο τελικά όλα αυτά οικοδομούν την κοινή αρχιτεκτονική των συστημάτων μας από μία υψηλού επιπέδου άποψη.

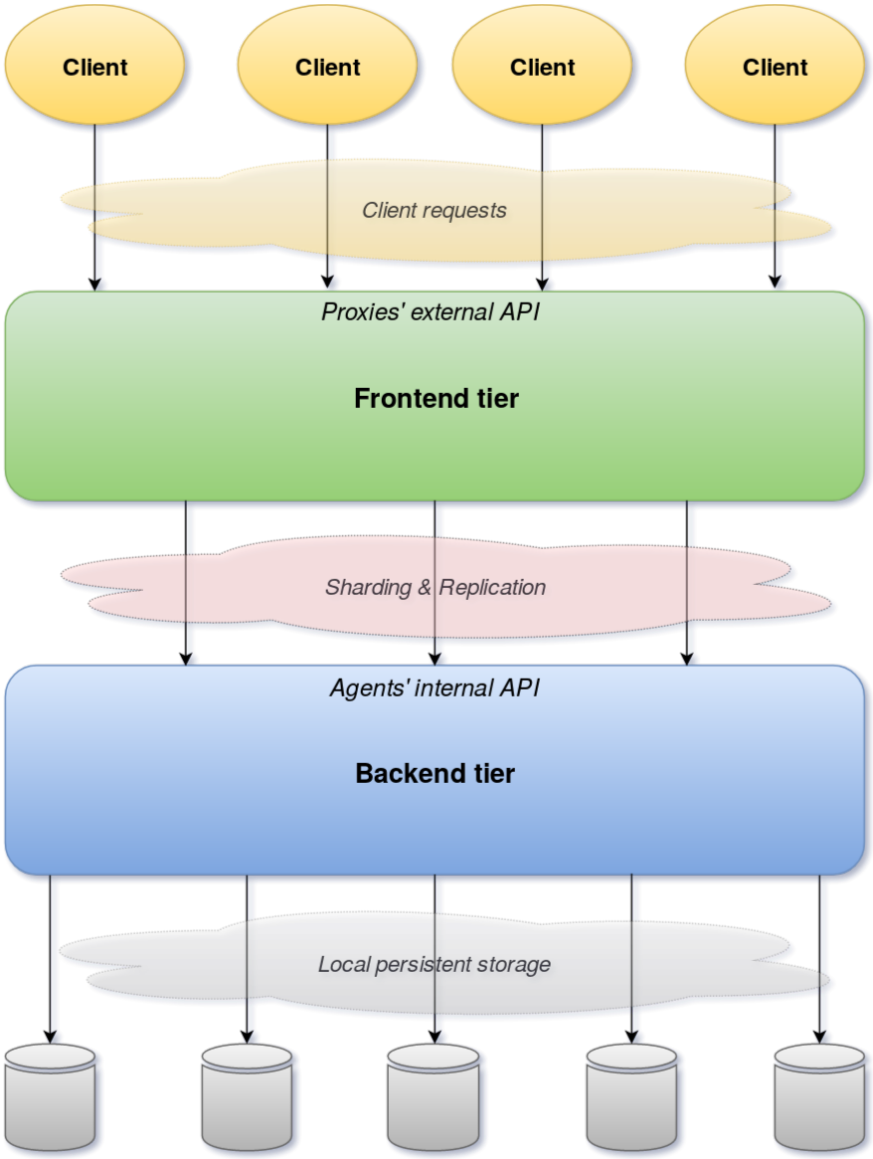
Όπως πρόκειται να γίνει σαφές, και ήταν αναμενόμενο, η επιρροή της Αρχιτεκτονικής Μικροϋπηρεσιών στη σχεδίασή μας είναι προφανής. Η προτεινόμενη αρχιτεκτονική είναι "πολυεπίπεδη": υπάρχει ένα "εμπρόσθιο επίπεδο" και ένα "οπίσθιο επίπεδο" [61]. Σε κάθε επίπεδο εκτελείται ένα μεταβλητό πλήθος στιγμιοτύπων. Τα στιγμιότυπα του εμπρόσθιου επιπέδου ονομάζονται "Διαμεσολαβητές", ενώ αυτά του οπίσθιου επιπέδο ονομάζονται "Πράκτορες", και στο εξής οι όροι "εμπρόσθιο επίπεδο" με "Διαμεσολαβητές" και "οπίσθιο επίπεδο" με "Πράκτορες" μπορεί να χρησιμοποιούνται ως εναλλακτικές εκφράσεις της ίδιας συνήθως αυτής έννοιας.

Το Σχήμα 4 απεικονίζει μία υψηλού επιπέδου άποψη της προτεινόμενης κοινής αρχιτεκτονικής των συστημάτων, με τρόπο ανεξάρτητο των επιλεγμένων τεχνικών τεμαχισμού και αυτόματης δημιουργίας αντιγράφων που τελικά χρησιμοποιεί το κάθε σύστημα. Στη συνέχεια, περιγράφουμε αφαιρετικά τις αρμοδιότητες των βασικών συστατικών στοιχείων.

Διαμεσολαβητές — το εμπρόσθιο επίπεδο

Το εμπρόσθιο επίπεδο έχει κυρίως δύο αρμοδιότητες. Κατ'αρχάς, είναι αυτό που επικοινωνεί με τους πελάτες: είναι το πρώτο συστατικό στοιχείο του συστήματος που λαμβάνει τα αιτήματά τους και αυτό που τελικά τους δίνει την τελική απάντηση. Η εξωτερική Διεπαφή Προγραμματισμού Εφαρμογών του συστήματος, η οποία σκιαγραφήθηκε νωρίτερα και μπορεί να βασίζεται είτε στο "REST" είτε στο "RPC", υλοποιείται και εκτίθεται από κάθε Διαμεσολαβητή. Η δεύτερη αρμοδιότητα των Διαμεσολαβητών είναι να λειτουργούν ως διαμεσολαβητές – εξ ου και το όνομά τους – για τα εισερχόμενα αιτήματα πελατών, προωθώντας τα στους κατάλληλους κάθε φορά Πράκτορες, και συναθροίζοντας τις απαντήσεις των τελευταίων προτού παράγουν την τελική απάντηση για τους πελάτες. Η απόφαση σχετικά με το ποιοι Πράκτορες πρέπει να λάβουν ποια προωθημένα αιτήματα πελατών καθορίζεται από την τεχνική

[61] Οι όροι "εμπρόσθιο επίπεδο" και "οπίσθιο επίπεδο" αναφέρονται στις έννοιες που εκφράζουν οι αγγλικοί όροι "frontend tier" και "backend tier".



Σχήμα 4: *Υψηλού επιπέδου απεικόνιση της προτεινόμενης αρχιτεκτονικής, η οποία είναι κοινή και στα δύο συστήματα.*

τεμαχισμού που χρησιμοποιείται από όλο το σύστημα, το οποίο κάνει τον ρόλο των Διαμεσολαβητών ιδιαιτέρως σημαντικό για τη διαδικασία του τεμαχισμού. Επιπροσθέτως, αφού οι Πράκτορες στους οποίους ένα αίτημα πελάτη πρέπει να προωθηθεί, μπορεί να είναι περισσότεροι του ενός, υποννοείται ότι τμήμα της λογικής του αλγορίθμου αυτόματης δημιουργίας αντιγράφων εκτελείται, και πάλι, στο εμπρόσθιο επίπεδο. Αυτό είναι εξάλλου αναμενόμενο, αφού συνήθως το σχήμα δημιουργίας αντιγράφων είναι βαθιά αλληλένδετο με την εκάστοτε μέθοδο τεμαχισμού.

Ένα σπουδαίο χαρακτηριστικό του εμπρόσθιου επιπέδου είναι ότι οι Διαμεσολαβητές δεν έχουν μόνιμη κατάσταση. Εφόσον δεν είναι οι ίδιοι που αποθηκεύουν τα δεδο-

μένα, δεν υπάρχει καμία πληροφορία κατάστασης που πρέπει να αποθηκευτεί με μόνιμο τρόπο. Η "κατάσταση" ενός Διαμεσολαβητή περιορίζεται στα τρέχοντα αιτήματα πελατών που πρέπει να απαντηθούν κάθε στιγμή, και η οποία είναι, προφανώς, πρόσκαιρη. Αυτού του είδους η πληροφορία κατάστασης αποθηκεύεται προσωρινά στη μνήμη μέχρι το αίτημα να απαντηθεί, χωρίς καμία ανάγκη πρόσβασης σε υλικό αποθήκευσης. Κατά συνέπεια, αποτυχίες δίσκων δεν μπορούν ποτέ να επιφέρουν προβλήματα στη λειτουργία του εμπρόσθιου επιπέδου του συστήματος, υπό οποιεσδήποτε συνθήκες.

Η έλλειψη ανάγκης για αποθήκευση μόνιμης κατάστασης που διέπει τη σχεδίαση των Διαμεσολαβητών, παρέχει ένα πολύ σημαντικό πλεονέκτημα σχετικό με την κλιμακωσιμότητα και την ελαστικότητά τους: το εμπρόσθιο επίπεδο μπορεί με τετριμμένο να κλιμακωθεί τόσο καθέτως όσο και οριζοντίως. Οι Διαμεσολαβητές κλιμακώνονται καθέτως – μέχρι κάποιο όριο, φυσικά – αφού το μόνο που χρειάζονται είναι υπολογιστική δύναμη και μνήμη. Όντας πολυνηματικοί, έτσι ώστε να τους είναι δυνατό να χειρίζονται πολλαπλά αιτήματα πελατών εν παραλλήλω[62], οι Διαμεσολαβητές είναι καθέτως κλιμακώσιμοι αφού η "πρσθήκη" όσο περισσότερων επεξεργαστικών πυρήνων και όσης περισσότερης χωρητικότητας μνήμης DRAM γίνεται στους φυσικούς κόμβους στους οποίους αυτοί εκτελούνται, μεταφράζεται ευθέως σε αυξημένη ικανότητα χειρισμού μεγαλύτερου πλήθους αιτημάτων πελατών εν παραλλήλω.

Το εμπρόσθιο τμήμα είναι, ακόμα, κλιμακώσιμο οριζοντίως: η εκτέλεση του απαιτούμενου πλήθους Διαμεσολαβητών σε διαφορετικούς κόμβους της υποκείμενης συστοιχίας αρκεί για τον σκοπό αυτό. Η έλλειψη της ανάγκης για διατήρηση μόνιμης κατάστασης των Διαμεσολαβητών εξασφαλίζει ότι αυτοί δεν έχουν ούτε ταυτότητα, ούτε προηγούμενη κατάσταση που να απαιτείται να βρουν και να συγχρονίσουν κατά την εκκίνησή τους. Επομένως, η διαδικασία οριζόντιας κλιμάκωσής τους είναι τετριμμένη: ο διαχειριστής του συστήματος μπορεί να δημιουργήσει ή να τερματίσει όσες σχετικές διεργασίες επιθυμεί. Μάλιστα, αυτό μπορεί να συμβεί οποιαδήποτε στιγμή κατά την λειτουργία του συστήματος, λόγου χάριν βάσει του τρέχοντος φόρτου του συστήματος, διασφαλίζοντας έτσι απόλυτη ελαστικότητα για το εμπρόσθιο τμήμα του συστήματος. Διασφαλίζεται επίσης ότι οι Διαμεσολαβητές είναι ανεκτικοί ως προς αποτυχίες κόμβων, με την έννοια ότι, παρότι κάποιοι από αυτούς μπορεί να εξαφανίζονται

[62] Μάλιστα, η υλοποίηση χρησιμοποιεί "ελαφριούς" μηχανισμούς νημάτων, όπως τα "goroutines" της γλώσσας Go και τα "greenlets" της γλώσσας Python, όποτε είναι εφικτό.

απροειδοποίητα, και συνεπώς τα τρέχοντα αναπάντητα αιτήματα πελατών που χειρίζονταν να πρέπει να επανεκκινηθούν, τέτοιες αποτυχίες δεν επηρεάζουν άλλα συστατικά στοιχεία του συστήματος, ούτε η άμεση επανεκκίνηση της διεργασίας σε οποιονδήποτε διαθέσιμο κόμβο της συστοιχίας περιορίζεται από ή επιφέρει κάποιο πρόβλημα στη σχεδίαση του συστήματος. Για τους λόγους αυτούς, παρότι οι Διαμεσολαβητές είναι ένα πολύ σημαντικό τμήμα του συστήματός μας, και παρότι είναι κατά κανόνα εντός της "διαδρομής των δεδομένων" μέχρι την αποθήκευσή τους, δεν ανεμένεται ποτέ να αποτελέσουν σημείο συμφόρησης για το σύστημά μας, ούτε να προκαλέσουν πρόβλημα στις επιδόσεις του γενικότερα, χάριν στον τρόπο με τον οποίον έχουν σχεδιαστεί.

Πράκτορες — το οπίσθιο επίπεδο

Η κύρια αρμοδιότητα του οπίσθιου επιπέδου είναι η μόνιμη αποθήκευση των μεγάλων δυαδικών αντικειμένων. Είναι αυτό το επίπεδο στο οποίο πρέπει να διασφαλίζεται η "μονιμότητα", και στο οποίο πρέπει ναι υλοποιηθεί οποιασδήποτε μορφής εσωτερική ευρετηριοποίηση καθώς και η ικανότητα του συστήματος να διαχειρίζεται ενδεχόμενες ταυτόχρονες πράξεις πάνω στα αποθηκευμένα αντικείμενα – είναι, με άλλα λόγια, το επίπεδο στο οποίο βρίσκεται η "μηχανή αποθήκευσης". Οι Πράκτορες υλοποιούν μία απλή, ξεχωριστή Διεπαφή Προγραμματισμού Εφαρμογών, η οποία δεν εκτίθεται εξωτερικά προς στους πελάτες, αλλά είναι προσβάσιμη από τους Διαμεσολαβητές, και η οποία μπορεί, και αυτή, να βασίζεται είτε στο "REST" είτε στο "RPC". Όταν οι Διαμεσολαβητές παραλαμβάνουν αιτήματα πελατών και αποφασίζουν σε ποιους Πράκτορες πρέπει να τα προωθήσουν, αυτή η προώθηση λαμβάνει χώρα εκτελώντας "πράξεις" που ορίζονται σε αυτή την εσωτερική Διεπαφή Προγραμματισμού Εφαρμογών, και η οποία υλοποιείται και εκτίθεται από κάθε Πράκτορα.

Το γεγονός ότι οι Πράκτορες είναι υπεύθυνοι για την αποθήκευση των δεδομένων είναι που τους κάνει από τη φύση τους να έχουν ανάγκη για διατήρηση μόνιμης κατάστασης, και αυτό είναι, ως συνήθως, μία επιβάρυνση. Κατ᾽αρχάς, η κατάσταση των Πρακτόρων πρέπει να αποθηκεύεται με μόνιμο τρόπο, και συνεπώς αυτοί εξαρτώνται σε μεγάλο βαθμό από την υποκείμενη υποδομή αποθηκευτικού υλικού, και άρα, εκτός των δικτυακών και των υπόλοιπων ειδών αποτυχιών που μπορεί να καταστήσουν έναν κόμβο μη-λειτουργικό, οι Πράκτορες είναι επιπροσθέτως ευάλωτοι σε σφάλματα και

αποτυχίες των δίσκων. Δεδομένου, λοιπόν, ότι έχουν ανάγκη για διατήρηση μόνιμης κατάστασης, η διαδικασία επαναφοράς τους μετά από αποτυχία δεν είναι όσο απλή είναι η αντίστοιχη διαδικασία των Διαμεσολαβητών, διότι οι Πράκτορες πρέπει με κάποιον τρόπο να "κουβαλήσουν", να μεταφέρουν μαζί τους τα δεδομένα για τα οποία είναι υπεύθυνοι, τα οποία μπορεί να είναι της τάξεως των TBs. Το γεγονός της "εξαφάνισης" ενός Πράκτορα, ιδίως όταν δεν έχει εγκατασταθεί αλγόριθμος αυτοματοποιημένης δημιουργίας αντιγράφων, ουσιαστικά μεταφράζεται ως μη-διαθεσιμότητα ενός τμήματος των αποθηκευμένων δεδομένων, δηλαδή σε συμπεριφορά απαράδεκτη για οποιουδήποτε είδους σύστημα αποθήκευσης.

Η εγκατάσταση κάποιου αλγορίθμου αυτοματοποιημένης δημιουργίας αντιγράφων τωόντι συνιστά ένα βήμα προς την βελτίωση της προκείμενης κατάστασης. Και πάλι, όμως, ακόμα και όταν τα αντικείμενα αποθηκεύονται σε πολλά αντίγραφα μεταξύ πολλαπλών Πρακτόρων, στην περίπτωση κάποιας αποτυχίας κόμβου κατά την οποία ο Πράκτορας πρέπει να επανεκκινηθεί σε έναν άλλο κόμβο της συστοιχίας, πρέπει με κάποιον τρόπο να αποκτήσει ξανά τα δεδομένα για τα οποία είναι υπεύθυνος, όσο ταχύτερα γίνεται, ώστε να είναι και πάλι λειτουργικός. Τα δεδομένα αποκτώνται ξανά με τη συνδρομή των υπόλοιπων Πρακτόρων που αποθηκεύουν αντίγραφα των ίδιων δεδομένων. Κατά τη διάρκεια της περιόδου κατά την οποία το σύνολο των αποθηκευμένων δεδομένων του Πράκτορα δεν είναι σε "έτοιμη κατάσταση", δεν πάσχει μόνο η διαθεσιμότητα ενός τμήματος των δεδομένων του τεμαχίου, αλλά είναι και αυξημένος ο κίνδυνος ολοκληρωτικής απώλειας των δεδομένων: έχοντας παράγοντα αντιγραφής, έστω k, αν τύχει όλοι οι k Πράκτορες που αποθηκεύουν αντίγραφα των αντικειμένων ενός τεμαχίου να αποτύχουν, κανένας από αυτούς δεν θα μπορεί να τα επανακτήσει κατά την επανεκκίνησή του στη συστοιχία. Γενικά, υπό του τρέχοντος μοντέλου αποτυχιών που υιοθετούμε, παράγοντας αντιγραφής k εγγυάται ανοχή σε $k-1$ σφάλματα, δεδομένης μιας σωστής σχεδίασης. Με άλλα λόγια, δεν μπορούν πάνω από $k-1$ Πράκτορες να είναι ταυτόχρονα σε κατάσταση αποτυχίας δίχως συνεπακόλουθη απώλεια δεδομένων. Οποιαδήποτε στιγμή, όσο περισσότεροι Πράκτορες είναι σε "ανέτοιμη κατάσταση", τόσο περισσότερο πάσχει η διαθεσιμότητα των δεδομένων στα τεμάχια που επηρεάζονται, και τόσο λιγότερες επιπρόσθετες αποτυχίες μπορούν να γίνουν ανεκτές στο ίδιο χρονικό διάστημα.

Υπάρχουν πολλοί τρόποι να υλοποιηθεί ο μηχανισμός δημιουργίας αντιγράφων ενός κατανεμημένου συστήματος αποθήκευσης. Κατ᾽αρχάς, το μοντέλο της δημιουργίας

αντιγράφων μπορεί να είναι ενεργητικό ή παθητικό. Σύμφωνα με το παθητικό μοντέλο, το εμπρόσθιο επίπεδο αλληλεπιδρά με έναν μοναδικό πρωτεύοντα διαχειριστή αντιγράφων, έναν "κύριο". Αυτός είναι υπεύθυνος για την απάντηση των εισερχόμενων αιτημάτων για το τεμάχιο εκ μέρους του οπίσθιου επιπέδου, καθώς και για την ενημέρωση ενός πλήθους δευτερευόντων διαχειριστών αντιγράφων, ή "σκλάβων", σχετικά με τροποποιήσεις στην αποθηκευμένη κατάσταση. Στην περίπτωση αποτυχίας του πρωτεύοντος, ένας εκ των δευτερευόντων προάγεται νέος πρωτεύων. Προσκολλώντας στις αρχές της "αντιγραφής μηχανής καταστάσεων", ο μηχανισμός αυτοματοποιημένης δημιουργίας αντιγράφων στο σύστημά μας βασίζεται στο προαναφερθέν ενεργητικό μοντέλο. Σύμφωνα με αυτό, δεν υπάρχει μοναδικός πρωτεύων διαχειριστής για κάθε αποθηκευμένο αντικείμενο: όλοι οι κάτοχοι αντιγράφων, δηλαδή εν προκειμένω οι Πράκτορες, είναι ισότιμοι. Το εμπρόσθιο τμήμα προωθεί τα αιτήματα των πελατών σε όλους αυτούς τους Πράκτορες, και καθένας τους τα εκτελεί ανεξάρτητα. Τα αποτελέσματα των εκτελέσεων αυτών συγκεντρώνονται στους Διαμεσολαβητές, οι οποίοι αποφασίζουν βάσει αυτών ποια θα είναι και η τελική απάντηση του συστήματος σε κάθε πελάτη.

Κατά συνέπεια, μία από τις πλέον σημαντικές – και μη τετριμμένες ως προς την ευκολία μιας ορθής σχεδίασης και υλοποίησής τους – αρμοδιότητες ενός Πράκτορα είναι η συμβολή του στην συνολική αντοχή του συστήματος σε σφάλματα και αποτυχίες, παρά την εγγενή του ανάγκη για διατήρηση μόνιμης κατάστασης. Ένας Πράκτορας πρέπει να είναι ενήμερος για τις περισσοτερες πτυχές των τεχνικών τεμαχισμού και αυτοματοποιημένης δημιουργίας αντιγράφων που χρησιμοποιούνται στο σύστημα. Πρέπει να ξέρει τουλάχιστον τον ακριβή αριθμό του συνόλου των τεμαχίων του συστήματος κάθε στιγμή, καθώς και τους υπόλοιπους Πράκτορες που είναι υπεύθυνοι για την αποθήκευση των δεδομένων στο τεμάχιο – ή στα τεμάχια – για τα οποία είναι υπεύθυνος και ο ίδιος. Πρέπει, ακόμα, να είναι σε θέση να επικοινωνήσει μαζί τους κατά την επαναφορά του, ώστε να καταφέρει να ανακτήσει τα δεδομένα για τα οποία είναι υπεύθυνος, πιθανώς χρησιμοποιώντας κάποια τεχνική μαζικής μεταφοράς εφόσον απαιτείται, αλλά και να είναι διαθέσιμος για να επικοινωνήσουν μαζί του άλλοι Πράκτορες κατά την δική τους επαναφορά, για τον ίδιο λόγο.

Όταν μία συστοιχία από Πράκτορες εκτελείται για πρώτη φορά, λέμε ότι είναι στη "φάση Εκκίνησης". Ένας Πράκτορας που όταν ξεκινά τη λειτουργία του η συστοιχία Πρακτόρων στην οποία ανήκει δεν είναι στη φάση Εκκίνησης, μπορεί μόνο να επα-

ναφέρεται από αποτυχία. Στην περίπτωση αυτή, όπως έχουμε ήδη έξηγήσει, επιχειρεί να ανακτήσει τα δεδομένα που θα έπρεπε να διατηρεί, το ταχύτερο δυνατόν. Ονομάζουμε αυτή τη διαδικασία "συγχρονισμό δεδομένων εκκίνησης" ή απλά "συγχρονισμό εκκίνησης" για συντομία. Συνήθως, λέμε ότι ένας Πράκτορας κατά τη διάρκεια της διαδικασίας συγχρονισμού εκκίνησης των δεδομένων του είναι στη "φάση Συγχρονισμού Εκκίνησης". Όταν η διαδικασία αυτή φτάσει εις πέρας, ο Πράκτορας συνήθως μεταβαίνει στη "φάση Ετοιμότητας", το οποίο σημαίνει ότι τα αποθηκευμένα δεδομένα του είναι πλήρως ενημερωμένα και ο ίδιος είναι έτοιμος να εξυπηρετήσει οποιοδήποτε προωθημένο από τους Διαμεσολαβητές αίτημα. Σημειώνεται ότι το δεύτερο εκ των δύο συστημάτων μας, έχει αρκετές περισσότερες "φάσεις" στις οποίες μεταβαίνει, είτε ο Πράκτορας, είτε η συστοιχία Πρακτόρων, όμως σ'αυτές θα αναφερθούμε αργότερα.

Κατά τη διάρκεια του συγρονισμού εκκίνησης, η μεταφορά των δεδομένων πρέπει να πληροί ορισμένες προϋποθέσεις. Αρχικά, πρέπει να λαμβάνει χώρα με κάποια αξιόπιστη μέθοδο, και συνεπώς, όπως συζητήθηκε νωρίτερα, το TCP είναι η καταλληλότερη επιλογή πρωτοκόλλου μεταφοράς. Επιπλέον, πρέπει να συνυπολογίσουμε ότι ένας Πράκτορας που επαναφέρεται από σφάλμα, θα μπορούσε υπό κανονικές συνθήκες να έχει ακόμα κάποια από τα αποθηκευμένα δεδομένα του – πιθανώς και όλα – αν, παραδείγματος χάριν, για οποιονδήποτε λόγο επαναχρονοδρομολογούνταν στον ίδιο φυσικό κόμβο κατά την επαναφορά του. Χρειαζόμαστε, λοιπόν, έναν "ευφυή" τρόπο να καθορίζουμε το τμήμα εκείνο των δεδομένων που πράγματι χρειάζεται συγχρονισμό με τους άλλους Πράκτορες έναντι εκείνου που υπάρχει ήδη αποθηκευμένο και ακέραιο κατά την εκκίνηση.

Τέλος, πρέπει να αποφασιστεί η κατεύθυνση της ροής των δεδομένων κατά τη μεταφορά. Το "μοντέλο έλξης" καθορίζει ότι η μεταφορά των δεδομένων είναι αρμοδιότητα του επαναφερόμενου Πράκτορα: είναι αυτός που κατά την επαναφορά του πρέπει να ενεργήσει, ρωτώντας τους υπόλοιπους Πράκτορες για τα τμήματα των δεδομένων για τα οποία είναι υπεύθυνος, και να δράσει ώστε να λάβει – να "έλξει" – όσα δεν έχει. Το "μοντέλο ώθησης" καθορίζει ότι η μεταφορά των δεδομένων είναι αρμοδιότητα των υγειών και ενημερωμένων Πρακτόρων: είναι εκείνοι που πρέπει να ελέγχουν για εκκινούμενους Πράκτορες, ρωτώντας τους για τα τμήματα των δεδομένων για τα οποία είναι υπεύθυνοι, και να δράσουν, όποτε κρίνεται αναγκαίο, αποστέλλοντας – "ωθώντας" – όσα τους λείπουν. Στην περίπτωσή μας, επιλέχθηκε η πρώτη εκ των δύο

ανωτέρω συλλογιστικών, αφού φαίνεται να ταιριάζει πιο φυσικά στην αρχιτεκτονική των συστημάτων μας: από κάθε Πράκτορα υπάρχει η απαίτηση να δέχεται αιτήματα συγχρονισμού από επαναφερόμενους Πράκτορες που ζητούν να έλξουν δεδομένα του που τους ενδιαφέρουν, και να τους εξυπηρετεί όσο καλύτερα δύναται. Όπως έχουμε προαναφέρει, ο σχεδιασμός και η υλοποίηση ειδικού πρωτοκόλλου επιπέδου εφαρμογής με όλα αυτά τα χαρακτηριστικά, το οποίο να είναι και ορθό αλλά και αποδοτικό, μολονότι αποτελεί την καλύτερη λύση, δεν είναι τετριμμένη διαδικασία, και σίγουρα δεν εμπίπτει στους βασικούς στόχους της παρούσας εργασίας. Γι᾽αυτόν το λόγο, επιλέγουμε να χρησιμοποιήσουμε ένα από τα πολλά υπάρχοντα και δοκιμασμένα πρωτόκολλα, το οποίο έχει μελετηθεί διεξοδικά και έχει αποδείξει την αξία του ακόμα και σε περιβάλλοντα παραγωγής τις τελευταίες δεκαετίες. Πρόκειται για το πρωτόκολλο “rsync”, το οποίο φαίνεται, με την κατάλληλη ρύθμισή του, να ικανοποιεί τις περισσότερες εκ των απαιτήσεών μας.

Kubernetes

Σ᾽αυτό το σημείο, είμαστε πλέον σε θέση να αντιστοιχίσουμε αφαιρετικά κάποια από τα συστατικά στοιχεία και τη λειτουργικότητα της κοινής αρχιτεκτονικής των συστημάτων μας σε στοιχεία της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes.

Το εμπρόσθιο επίπεδο απαρτίζεται από ένα μεταβλητό πλήθος Διαμεσολαβητών, και απαιτεί μόνο ένα μικρό υποσύνολο των παρεχόμενων χαρακτηριστικών του Kubernetes. Συγκεκριμένα, απαιτεί την βασικότερη λειτουργία του Kubernetes: να παρακολουθεί την τρέχουσα κατάσταση κάθε Διαμεσολαβητή, και να διασφαλίζει ότι καθένας τους εκτελείται και εξυπηρετεί αιτήματα πελατών, και βέβαια να παρεμβαίνει σε περιπτώσεις αποτυχιών, θέτοντας σε εκτέλεση νέα στιγμιότυπα Διαμεσολαβητών σε διαθέσιμους και κατάλληλους κόμβους της συστοιχίας. Η μονάδα “deployment” του Kubernetes είναι το “Pod”· ευθέως, λοιπόν, αντιστοιχίζουμε κάθε εκτελεστέο στιγμιότυπο Διαμεσολαβητή με ένα Pod αποτελούμενο από ένα περιέκτη, δηλαδή ένα Pod που περιλαμβάνει ένα μόνο περιέκτη Docker, στον οποίον εκτελείται μία μόνο διεργασία: το εκτελέσιμο αρχείο του Διαμεσολαβητή.

Υπάρχουν περισσότερες από μία διαθέσιμες επιλογές για το υψηλότερου επιπέδου στοιχείο της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes που παρακολουθεί τα Pods των Διαμεσολαβητών. Αναλογιζόμενοι ότι οι Διαμεσολαβητές είναι

πλήρως απαλλαγμένοι από την ανάγκη για διατήρηση μόνιμης κατάστασης, και ότι απλά πρέπει να εξυπηρετούν αιτήματα πελατών για πάντα – μέχρι είτε να διακοπούν είτε να αποτύχουν – καταλήγουμε ότι οι καλύτερες επιλογές μας είναι τα `Deployment` και `DaemonSet`. Η επιλογή μεταξύ των δύο εξαρτάται σε μεγάλο βαθμό και από την υπόλοιπη σύσταση της συστοιχίας, καθώς και από τις ανάγκες της εκάστοτε περίπτωσης. Η κύρια διαφορά τους είναι ότι το `DaemonSet` διασφαλίζει ότι όλοι οι κόμβοι της συστοιχίας του Kubernetes θα εκτελούν πάντα ένα στιγμιότυπο του Pod, κάθε στιγμή, ενώ το `Deployment` διασφαλίζει ότι ένα συγκεκριμένο πλήθος από Pods εκτελούνται κάθε στιγμή σε οποιουσδήποτε κόμβους της συστοιχίας. Και οι δύο αυτές επιλογές συμμορφώνονται σε διαχειριστικές ρυθμίσεις σχετικές με την δρομολόγησή τους από το Kubernetes [180], ώστε κόμβοι να μπορούν να συμπεριληφθούν ή να αποκλειστούν από το αντίστοιχο σύνολο υποψήφιων κόμβων. Είναι προφανές, ωστόσο, ότι τις περισσότερες φορές το `Deployment` είναι η πιο ευέλικτη επιλογή, αφού καθιστά την οριζόντια κλιμάκωση του επιπέδου όσο ευκολότερη γίνεται, θέτοντας απλά το επιθυμητό πλήθος εκτελούμενων στιγμιότυπων.

Στο οπίσθιο επίπεδο, το οποίο αποτελείται από ένα μεταβλητό πλήθος Πρακτόρων, οι απαιτήσεις είναι αυστηρότερες. Κατ᾽αρχάς, τα βασικότερα χαρακτηριστικά του Kubernetes είναι ακόμα απαιτούμενα: το Kubernetes πρέπει να παρακολουθεί την τρέχουσα κατάσταση κάθε Πράκτορα και να παρεμβαίνει όποτε κάτι δεν είναι όπως θα έπρεπε, επανεκκινώντας αποτυχημένους Πράκτορες σε εναλλακτικούς κατάλληλους και διαθέσιμους κόμβους της συστοιχίας. Αυτή τη φορά, όμως, και σε αντίθεση με την περίπτωση των Διαμεσολαβητών, αφού κάθε Πράκτορας είναι “δέσμιος” των δεδομένων που αποθηκεύει, και της αρμοδιότητάς του να τα διατηρεί αποθηκευμένα και ενημερωμένα, κάθε Πράκτορας μπορεί να ωφεληθεί από την ύπαρξη μίας ταυτότητας που είναι μοναδική μεταξύ των υπόλοιπων Πρακτόρων. Επιπροσθέτως, καθώς οι Πράκτορες μπορεί συχνά να πρέπει να επικοινωνήσουν μεταξύ τους ώστε να συγχρονίσουν τα αποθηκευμένα τους δεδομένα – για την ακρίβεια, ο καθένας τους με συγκεκριμένους άλλους – μπορούν ακόμα να επωφεληθούν διατηρώντας την μοναδική τους αυτή ταυτότητα συνδεδεμένη με μία σταθερή διεύθυνση δικτύου, ως μέσο επικοινωνίας με τους υπόλοιπους Πράκτορες οποιαδήποτε στιγμή, πλην φυσικά των διαστημάτων αποτυχίας τους. Όλα αυτά είναι χαρακτηριστικά που το Kubernetes μπορεί να παράσχει μέσω του `StatefulSet`, ένα αντικείμενο της Διεπαφής Προγραμματισμού

Εφαρμογών του, του οποίου η ύπαρξη έχει ως κύριο σκοπό την προκείμενη επιδίωξή μας: την εκτέλεση και διαχείριση εφαρμογών και συστημάτων που έχουν την ανάγκη διατήρησης μόνιμης κατάστασης στους κόμβους της συστοιχίας.

Παρόμοια με τα `Deployment` και `DaemonSet`, το `StatefulSet` είναι ένα υψηλότερου επιπέδου αντικείμενο της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes, το οποίο παρακολουθεί σύνολα από Pods. Όπως και στην περίπτωση των Διαμεσολαβητών, κάθε Πράκτορας περικλείεται σε ένα Pod. Αντίθετα όμως με την περίπτωση των Διαμεσολαβητών, αυτή τη φορά το Pod συμπεριλείει δύο περιέκτες Docker αντί για έναν. Στον πρώτο περιέκτη εκτελείται μία διεργασία: το εκτελέσιμο αρχείο του Πράκτορα, ενώ στον δεύτερο εκτελείται ως δαίμονας ένας διακομιστής rsync, με σκοπό να εξυπηρετεί αιτήματα συγχρονισμού δεδομένων εκκίνησης άλλων Πρακτόρων. Οι λεπτομέρειες σχετικά με τη χρήση των `StatefulSet` διαφέρουν μεταξύ των δύο συστημάτων, συνεπώς η συζήτηση προς το παρόν αναβάλλεται για κάποιο επόμενο σημείο της εργασίας.

Πρέπει να έχει γίνει ήδη εμφανές ότι η επικοινωνία τόσο των εκτελούμενων συστατικών στοιχείων της αρχιτεκτονικής του συστήματος μεταξύ τους, όσο και με τους πελάτες, συμβάλλει καθοριστικά στη φυσιολογική λειτουργία του. Παρ᾽ολ᾽αυτά, είναι ήδη γνωστό ότι το περιβάλλον των σύγχρονων κέντρων δεδομένων, τα οποία αποτελούνται από "ευρέως εμπορευματοποιημένες" υποδομές υλικού, καθιστά τους εκτελούμενους περιέκτες επιρρεπείς σε σφάλματα και αποτυχίες, ιδίως με την αύξηση των αναγκών και άρα του μεγέθους και της πολυπλοκότητας της συστοιχίας. Καθώς στιγμιότυπα του εμπρόσθιου και του οπίσθιου επιπέδου αποτυγχάνουν και επανεκκινούνται, σε διαφορετικούς πιθανώς κόμβους, πρέπει να τους είναι και πάλι δυνατή η επικοινωνία με άλλα στοιχεία του συστήματος, παρότι οι δικτυακές τοποθεσίες (διευθύνσεις IP και πόρτες) μπορεί να αλλάζουν διαρκώς. Το ίδιο ζήτημα αναφύεται και σε περιπτώσεις της οριζόντιας κλιμάκωσης, οπότε τέτοια στιγμιότυπα πρέπει να δημιουργούνται ή να πεθαίνουν ανά πάσα στιγμή. Αυτή είναι και η χονδροειδής περιγραφή του κοινού μεταξύ πολλών κατανεμημένων συστημάτων προβλήματος της "εξερεύνησης υπηρεσιών".

Το Kubernetes παρέχει μία αυτοματοποιημένη λύση για την εξερεύνηση υπηρεσιών, μέσω των αντικειμένων της Διεπαφής Προγραμματισμού Εφαρμογών του, τα `Service` και `Endpoint` [84]. Υποστηρίζονται από έναν υποκείμενο μηχανισμό, ο οποίος μπορεί

να ποικίλει από συστοιχία Kubernetes σε συστοιχία Kubernetes, και ο οποίος αναθέτει προσεκτικά εικονικές διευθύνσεις IP στα εκτελούμενα Pods, και οι οποίες έχουν νόημα μόνο εντός της συστοιχίας. Στη δική μας περίπτωση αυτός ο μηχανισμός είναι πάντα ένα δίκτυο επικάλυψης που δημιουργεί το εργαλείο `flannel` χρησιμοποιώντας δίκτυα `VXLAN`. Τα ίδια τα `Service` είναι ένας μηχανισμός που επικεντρώνεται σε ένα σύνολο από Pods, τα οποία επιλέγονται βάσει των `Label` και `Selector` αντικειμένων της Διεπαφής Προγραμματισμού του Kubernetes, και διευκολύνει την πρόσβαση σε αυτά απαλλάσσοντας από την ανάγκη πρότερης γνώσης της διεύθυνσης IP τους, η οποία μπορεί να αλλάζει δυναμικά. Έτσι και στην περίπτωσή μας, τα Pods ελέγχονται από τα υψηλότερου επιπέδου αντικείμενα της Διεπαφής Προγραμματισμού του Kubernetes που χρησιμοποιούνται, δηλαδή τα `Deployment` ή `DaemonSet` και `StatefulSet`, και εκθέτουν τις Διεπαφές Προγραμματισμού Εφαρμογών τους χρησιμοποιώντας τον μηχανισμό των `Service`.

Οι Διαμεσολαβητές πρέπει να είναι σε θέση να επικοινωνήσουν με τους πελάτες, οι οποίοι μπορεί να βρίσκονται εκτός συστοιχίας, πρέπει, δηλαδή, η Διεπαφή Προγραμματισμού Εφαρμογών τους να εκτίθεται “εξωτερικά”. Για το λόγο αυτόν, δύο είδη `Services` μπορούν να χρησιμοποιηθούν: τα `NodePort` και `LoadBalancer`. Η επιλογή μεταξύ των δύο εξαρτάται από την εκάστοτε ρύθμιση και σύνθεση της συστοιχίας και του περιβάλλοντος – μπορεί, λόγου χάριν, να χρησιμοποιείται κάποιος τρίτος πάροχος που παρέχει ιδιαίτερες υπηρεσίες. Στο εξής, υποθέτουμε για απλότητα και γενικότητα, ότι η Διεπαφή Προγραμματισμού Εφαρμογών του εμπρόσθιου επιπέδου των συστημάτων μας εκτίθεται εξωτερικά μέσω Kubernetes `Service` τύπου `NodePort`.

Οι Πράκτορες πρέπει να είναι σε θέση να επικοινωνήσουν με τους Διαμεσολαβητές μέσω της δικιάς τους Διεπαφής Προγραμματισμού Εφαρμογών προκειμένου να διαχειρίζονται τα προωθούμενα αιτήματα πελατών, καθώς και με άλλους Πράκτορες ώστε είτε να εκπέμψουν είτε να εξυπηρετήσουν αιτήματα συγχρονισμού δεδομένων εκκίνησης. Δεν υπάρχει, λοιπόν, καμία ανάγκη να επικοινωνήσουν με οποιαδήποτε οντότητα εκτός συστοιχίας. Αναλογιζόμενοι ότι το οπίσθιο τμήμα χρησιμοποιεί `StatefulSet`, τα οποία πάντα χρειάζονται ένα `Headless Service`, αφού δεν απαιτούν ούτε αυτοματοποιημένη εξισορρόπηση φόρτου ούτε μία ενιαία διεύθυνση IP για όλα τους, αρκεί αυτό το `Headless Service` ανά Πράκτορα για να καλύψει όλες τις ανάγκες επικοινωνίας τους με άλλα στοιχεία της σύνθεσης του συστήματος.

Τέλος, το Kubernetes είναι υπεύθυνο για την παροχή τοπικού μόνιμου χώρου αποθήκευσης στους Πράκτορες, είτε μέσω των `Volume` [186] είτε μέσω των `PersistentVolume` και `PersistentVolumeClaim` [187]. Εξαιτίας της εξαιρετικά ταχείας ανάπτυξης του Kubernetes, κατά τη συγγραφή της παρούσας εργασίας το Kubernetes υποστηρίζει – αν και η υλοποίηση δεν είναι ολοκληρωμένη και στην τελική της μορφή – την παροχή τοπικού μόνιμου χώρου αποθήκευσης στα Pods στατικά, μέσω `PersistentVolumes`, προσφέροντας πολλά πλεονεκτήματα στις εφαρμογές που τα χρειάζονται, αλλά και σημαντικούς περιορισμούς. Σε κάθε περίπτωση, η ανάλυσή τους δεν είναι σκοπός της παρούσας εργασίας, αφού κατά τη φάση της σχεδίαση και της υλοποίησης των συστημάτων μας, οι μοναδικές επιλογές παροχής τοπικού αποθηκευτικού χώρου ήταν πολύ περιορισμένες. Γι'αυτό το λόγο, καταλήξαμε να χρησιμοποιούμε έναν από τους απλούστερους τύπους Kubernetes `Volume`, το `emptyDir`.

Ένα `emptyDir` δημιουργείται όταν ένα Pod δρομολογείται προς εκτέλεση σε έναν κόμβο της συστοιχίας Kubernetes, και υπάρχει όσο το Pod εκτελείται στον κόμβο αυτό. Όπως υπονοείται από το όνομά του, αρχικά είναι άδειο, και όλοι οι περιέκτες εντός του Pod μπορούν να εγγράφουν και να διαβάζουν τα ίδια αρχεία στο `emptyDir`, ακόμα και αν έχει γίνει η προσάρτησή του σε διαφορετικό μονοπάτι του συστήματος αρχείου για τον καθένα. Όταν ένα Pod απομακρύνεται από τον αντίστοιχο κόμβο, για οποιονδήποτε λόγο, τα δεδομένα στο `emptyDir` διαγράφονται για πάντα, και αυτό μερικές φορές περιπλέκει την κατάσταση, υποβάλλοντάς της "τραχύτητα" και προσωρινότητα. Η παροχή τοπικού μόνιμου αποθηκευτικού μηχανισμού στα Pods των Πρακτόρων χρησιμοποιώντας τα νέα χαρακτηριστικά του Kubernetes αποτελεί ένα από τα βασικότερα σημεία της μελλοντικής επέκτασης που πρέπει να γίνει στα συστήματα που σχεδιάστηκαν.

Όπως έχει αναφερθεί επανειλημμένως, η φάση της σχεδίασης ήταν μια επαναληπτική διαδικασία που είχε ως αποτέλεσμα δύο ξεχωριστά συστήματα. Μέχρι τώρα, αναφορές "στο σύστημα" συνήθως αναφερόμασταν και στα δύο, αφού οι περισσότερες έννοιες που έχουν συζητηθεί έως τώρα, καθώς και η υψηλού επιπέδου αρχιτεκτονική, εφαρμόζονται και στα δύο. Καθώς, όμως, το καθένα τους τελικά υποστηρίζει διαφορετικά χαρακτηριστικά, και παρουσιάζει διαφορετικά πλεονεκτήματα και μειονεκτή-

ματα, ακολούθως θα αναφερθούμε σε καθένα ξεχωριστά.

MICAS

Το `MICAS`, το οποίο είναι ακρωνύμιο για το "Modulo-based Immutable Content-Addressable Storage", είναι το αποτέλεσμα της αρχικής μας προσέγγισης στη σχεδίαση του κατανεμημένου συστήματος αποθήκευσης, και το πρώτο από τα δύο συστήματα που παρουσιάζονται στα πλαίσια της παρούσας εργασίας.

Τεμαχισμός με Κατακερματισμό mod-N

Η μέθοδος τεμαχισμού που χρησιμοποιείται στο `MICAS` ονομάζεται "κατακερματισμός mod-N", και είναι αρκετά απλή σαν σκέψη. Ξεκινούμε επιλέγοντας την h, μία συνάρτηση κατακερματισμού N_h δυφίων [63]. Κατά τη δημιουργία ενός εισερχόμενου μεγάλου δυαδικού αντικειμένου, το περιεχόμενό του δίνεται ως είσοδος στη συνάρτηση κατακερματισμού, της οποίας η έξοδος είναι το κλειδί του νέου ζεύγους κλειδιού-τιμής που αποθηκεύεται για το νέο αντικείμενο.

Στη συνέχεια, έχοντας επιλέξει ένα στατικό πλήθος τεμαχίων, έστω N_s, στο οποίο κατατέμνεται το εύρος κλειδιών (ο επανατεμαχισμός συζητείται αργότερα), και έχοντάς τους αναθέσει από έναν ακέραιο αριθμό στο εύρος $[0, N_s - 1]$ ως μοναδικό αναγνωριστικό του καθενός, υπολογίζεται το τεμάχιο, S_i, στο οποίο ανατίθεται το αντικείμενο b_i, από τον εξής απλό υπολογισμό:

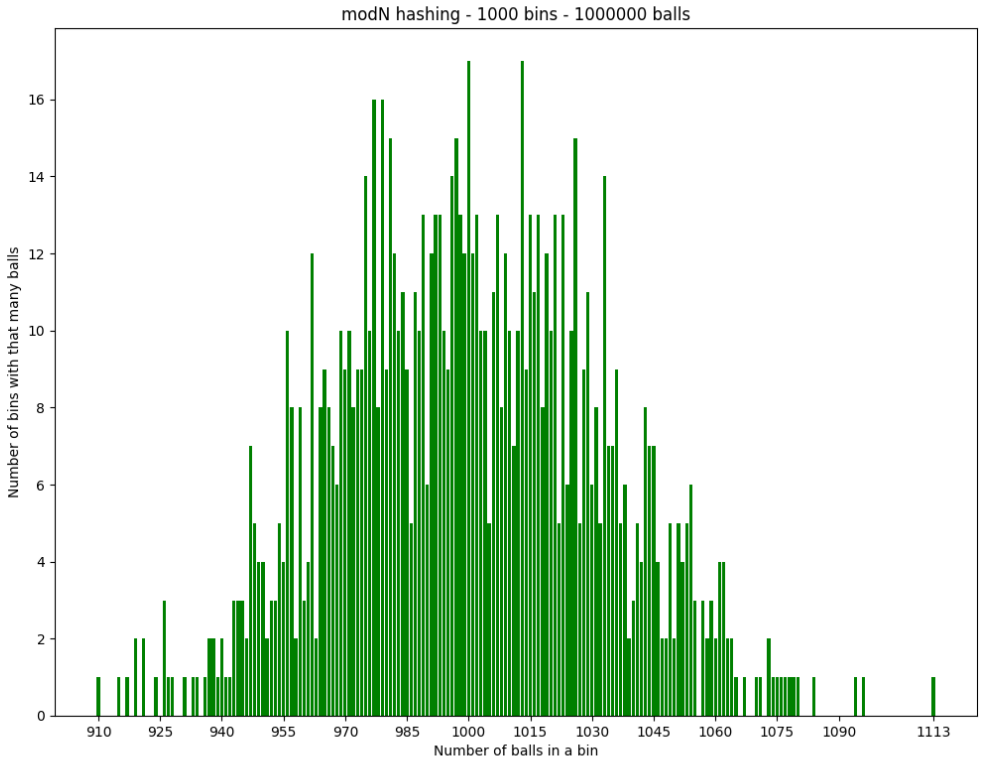
$$S_i = h(b_i) \bmod N_s \tag{5}$$

όπου το $h(b_i)$ αναπαριστά την εφαρμογή της συνάρτησης κατακερματισμού, h, στο περιεχόμενο του μεγάλου δυαδικού αντικειμένου, b_i, δηλαδή την αντίστοιχη σύνοψη κατακερματισμού. Στη χρήση της πράξης του υπολοίπου ακέραιας διαίρεσης οφείλεται, παρεμπιπτόντως, και το όνομα του συστήματος. Ως πράξη, το υπόλοιπο ακέραιας διαίρεσης, διατηρώντας τον διαιρέτη, d, σταθερό, μπορεί να θεωρηθεί ως ένας τρόπος ομαδοποίησης των διαιρετέων σε d ομάδες, οι οποίες περιέχουν τους διαιρετέους που

[63] Στο εξής, όποτε αναφερόμαστε στην συνάρτηση κατακερματισμού του `MICAS`, εννοείται ότι έχει επιλεχθεί κάποια τέτοια σύμφωνα με την ανάλυση που προηγήθηκε στις προηγούμενες ενότητες.

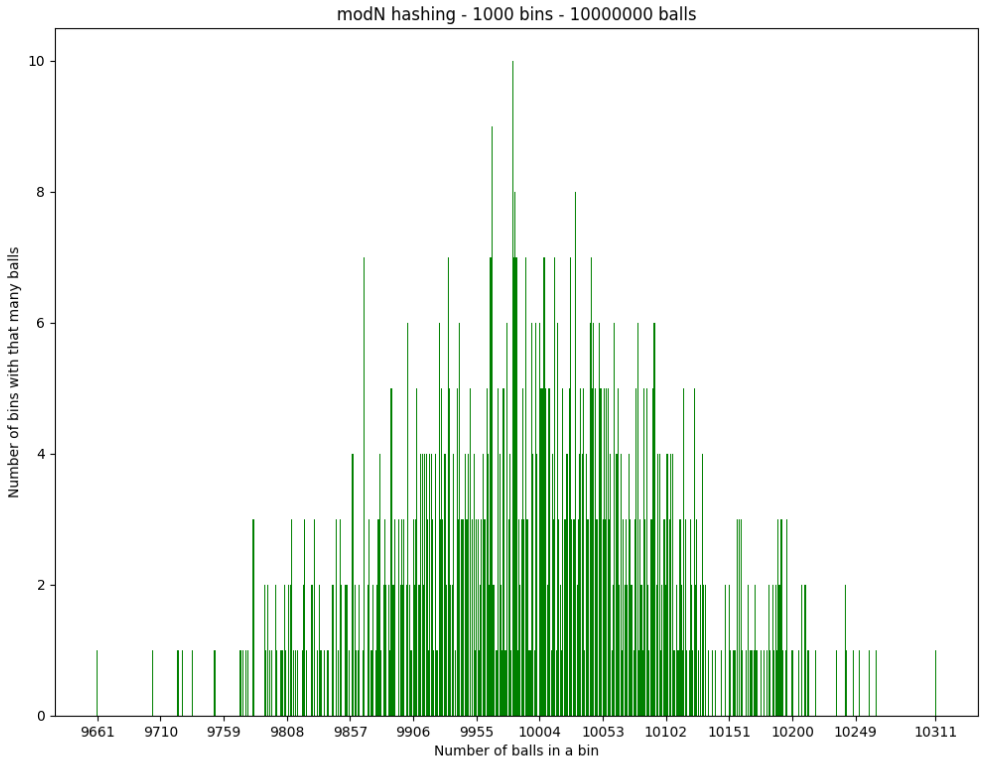
διαιρούμενοι με το d έχουν κοινό υπόλοιπο, και οι οποίες με τη σειρά τους μπορούν να αριθμηθούν βάσει του υπολοίπου αυτού με τους ακέραιους αριθμούς στο εύρος $[0,\ d-1]$.

Μία καλή αναλογία για τη μοντελοποίηση προβλημάτων εξισορρόπησης φόρτου, είναι η θεώρηση των τεμαχίων ως "κουβάδες", και των συνόψεων κατακερματισμού ως "μπάλες" [181, 182]. Έτσι, έχουμε συνολικά N_s κουβάδες και οι μπάλες διανέμονται "τυχαία" σ'αυτούς με τρόπο ώστε κάθε μπάλα να έχει πιθανότητα $\frac{1}{N_s}$ να μπει σε κάθε κουβά. Στην περίπτωσή μας, οι μπάλες, ως συνόψεις κατακερματισμού, έστω πλήθους n, μπορούν να θεωρηθούν ως ένα τυχαίο δείγμα του πληθυσμού όλων των ακεραίων στο εύρος κλειδιών $[0,\ 2^{N_h}-1]$. Διαισθητικά, αντιλαμβανόμαστε ότι καθένας εκ των N_s κουβάδων πληρώνεται με περίπου $\frac{n}{N_s}$ μπάλες, καθώς και ότι για μεγάλες τιμές του n, όταν η εξισορρόπηση φόρτου πράγματι έχει νόημα, οι μπάλες κατανέμονται στους κουβάδες με κανονικότητα γύρω από το $\frac{n}{N_s}$.



Σχήμα 5: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας κατακερματισμό mod-N για $N_s = 1000$ κουβάδες και $n = 10^6$ μπάλες. $\frac{n}{N_s} = 1000$ μπάλες ανά κουβά.*

Μία σειρά προσομοιώσεων Monte Carlo για διάφορες τιμές του n και σταθερό πλήθος κουβάδων, $N_s = 1000$, επιβεβαιώνει τη διαισθητική μας διαπίστωση. Στις γραφικές παραστάσεις των Σχημάτων 5 και 6, βλέπουμε τον τρόπο με τον οποίον έχουν κατανεμηθεί οι μπάλες σε καθέναν από τους $N_s = 1000$ κουβάδες στο τέλος κάθε προσομοίωσης. Ο οριζόντιος άξονας δείχνει τις τιμές του πλήθους μπαλών ανά κουβά, ενώ ο κατακόρυφος άξονας δείχνει το πλήθος των κουβάδων που έχει όσες μπάλες δείχνει ο οριζόντιος.



Σχήμα 6: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας κατακερματισμό mod-N για* $N_s = 1000$ *κουβάδες και* $n = 10^7$ *μπάλες.* $\frac{n}{N_s} = 10000$ *μπάλες ανά κουβά.*

Παρά τις αξιόλογες επιδόσεις του κατακερματισμού mod-N ως προς την εξισορρόπηση φόρτου, έχει ένα πολύ βασικό μειονέκτημα ως μέθοδος τεμαχισμού: το πλήθος των τεμαχίων καθορίζεται στατικά, και δεν υποστηρίζεται η μετέπειτα τροποποίησή του, δηλαδή ο επανατεμαχισμός. Έστω ότι κάτι τέτοιο υποστηριζόταν. Έχοντας διαμερίσει τα δεδομένα σε N_s τεμάχια, ας υποθέσουμε ότι επιθυμούμε να τροποποιήσουμε το πλήθος των τεμαχίων, σε N'_s, προκειμένου να κλιμακώσουμε οριζοντίως το σύστημα, είτε αυξάνοντας είτε μειώνοντας το πλήθος των τεμαχίων. Για τις περισσό-

τερες τιμές N'_s, βάσει της Equation 5, η συντριπτική πλειοψηφία των συνόψεων κατακερματισμού – και άρα τα αντικείμενα στα οποία αυτές έχουν ανατεθεί ως κλειδιά του αντίστοιχου ζεύγους κλειδιού-τιμής, ανήκουν πλέον σε διαφορετικό τεμάχιο από ό,τι πριν. Αυτό οδηγεί στην σχεδόν πλήρη αναδιάταξη όλων των δεδομένων μεταξύ των τεμαχίων – και άρα των κόμβων στους οποίους αυτά είναι μοιρασμένα. Στην ιδανική περίπτωση, θα ήταν απαραίτητο να μεταφερθούν μόνο το $\frac{N'_s - N_s}{N'_s}$ των δεδομένων όταν $N'_s > N_s$, ή το $\frac{N_s - N'_s}{N_s}$ των δεδομένων όταν $N'_s < N_s$, ώστε τα δεδομένα να καταλήξουν και πάλι ισομοιρασμένα μεταξύ των N'_s τεμαχίων· μάλιστα, το ποσοστό των δεδομένων που θα έπρεπε να μεταφερθούν μειώνεται όσο μειώνεται η τιμή $|N'_s - N_s|$, δηλαδή όσο μικρότερη είναι η κλιμάκωση του συστήματος – η οποία συχνά γίνεται, πράγματι, σε μικρά βήματα. Η "κακή" αυτή αναδιάταξη οφείλεται στην ίδια τη φύση της πράξης του υπολοίπου ακέραιας διαίρεσης, αφού αυτό πρέπει να επανυπολογιστεί για κάθε σύνοψη κατακερματισμού χρησιμοποιώντας διαφορετικό διαιρέτη.

Αφού ο επανατεμαχισμός δεν υποστηρίζεται, ο διαχειριστής του συστήματος θα πρέπει να ρυθμίσει το πλήθος των τεμαχίων βάσει της μέγιστης τιμής που αναμένεται να χρειαστεί να φτάσει. Όταν το πλήθος των τεμαχίων είναι μεγαλύτερο από το πλήθος των υποκείμενων κόμβων της συστοιχίας Kubernetes στην οποία το `MICAS` πρόκειται να εκτελεστεί, προφανώς κάποια τεμάχια θα χρειαστεί να τοποθετηθούν στον ίδιο κόμβο. Τεμαχίζοντας το εύρος κλειδιών εξαρχής βάσει του μέγιστου πλήθους τεμαχίων που αναμένεται να χρησιμοποιηθούν, η κλιμάκωση του συστήματος λαμβάνει χώρα απλά προσθέτοντας και αφαιρώντας κόμβους από τη συστοιχία, και τοποθετώντας–δρομολογώντας τα τεμάχια αυτά, και φυσικά όλα τα αντικείμενα που συμπερικλείονται, κάθε φορά πυκνότερα ή αραιότερα (ανάλογα με το είδος της κλιμάκωσης). Με άλλα λόγια, η χωρητικότητα του συστήματος αυξομειώνεται με την προσθαφαίρεση κόμβων της συστοιχίας, διατηρώντας ταυτόχρονα το πλήθος των τεμαχίων σταθερό, και μεταβάλλοντας μόνο την τοπολογία τους ως προς τους κόμβους αυτούς.

Αυτού του είδους η δυναμική προσαρμοστικότητα είναι που χαρακτηρίζει το `MICAS` ως ένα βαθμό ελαστικό . Η ελαστικότητα αυτή, όμως, περιορίζεται από την ρύθμιση του πλήθους των τεμαχίων στατικά, εξαρχής. Η σχεδίαση του δεύτερου συστήματος, του `RICAS`, είναι εκείνη που πρόκειται να μας απαλλάξει από το πρόβλημα της μειωμένης ελαστικότητας.

Αυτοματοποιημένη Δημιουργία Αντιγράφων

Όπως έχει επισημανθεί επανειλημμένως, τα συστήματα αποθήκευσης της παρούσας εργασίας έχουν ως στόχο την υψηλή διαθεσιμότητα. Για να επιτευχθεί αυτό, πρέπει να υπάρχει κάποιος μηχανισμός που να επιτρέπει τη δημιουργία και την ανάγνωση αντικειμένων ακόμα και όταν κάποιοι από τους κόμβους της συστοιχίας, στην οποία εκτελείται το σύστημα, έχουν αποτύχει. Πρέπει με άλλα λόγια το κάθε σύστημα, ως σύνολο επιμέρους συστατικών αρχιτεκτονικών στοιχείων, να είναι ανεκτικό σε σφάλματα και αποτυχίες. Η αυτοματοποιημένη δημιουργία αντιγράφων είναι μια τεχνική που εφαρμόζεται για να διασφαλιστεί ότι τα αντικείμενα δημιουργούνται σε πολλαπλούς κόμβους της υποκείμενης συστοιχίας, και αργότερα είναι δυνατή η ανάγνωσή τους από οποιουσδήποτε εξ αυτών.

Στην περίπτωση του `MICAS` και του κατακερματισμού mod-N, η συλλογιστική και η διαδικασία της αυτοματοποιημένης δημιουργίας αντιγράφων είναι πολύ απλή. Χρησιμοποιώντας την αναλογία της θεώρησης των τεμαχίων ως κουβάδες, η ιδέα της δημιουργίας αντιγράφων είναι η εξής: δεδομένου παράγοντα αντιγραφής k, μπορούν να διατηρούνται k αντίγραφα ολόκληρου του κουβά, ώστε συνολικά, για N_s τεμάχια να έχουμε $k \cdot N_s$ κουβάδες. Όποτε μια μπάλα (ένα αντικείμενο) πρέπει να τοποθετηθεί σε έναν κουβά, αντίγραφα της μπάλας τοποθετούνται σε όλα τα k αντίγραφα του κουβά στον οποίον αντιστοιχεί. Παρομοίως, όταν μία μπάλα πρέπει να αναζητηθεί σε κάποιον κουβά, είναι γνωστό ότι ένα αντίγραφό της πρέπει να έχει δημιουργηθεί σε καθένα από τα k αντίγραφα του κουβά στον οποίον έχει ανατεθεί, και συνεπώς μπορεί να ευρεθεί σε οποιοδήτε από αυτά.

Λεπτομέρειες για την υλοποίηση αυτής της λογικής καθώς και την ενσωμάτωσή της στην αρχιτεκτονική του `MICAS` θα γίνουν σαφείς στη συνέχεια.

Αρχιτεκτονική

Όπως έχει εξηγηθεί νωρίτερα, το `MICAS` αποτελείται από δύο επίπεδα: το εμπρόσθιο και το οπίσθιο.

Οι Διαμεσολαβητές στο εμπρόσθιο επίπεδο μπορούν να εκτελεστούν στους κόμβους της συστοιχίας Kubernetes χρησιμοποιώντας Pods, τα οποία ελέγχοντα είτε μέσω ενός `Deployment`, είτε μέσω ενός `DaemonSet`, όπως έχει αναφερθεί προηγουμένως. Στο

εξής, το Deployment είναι το προτιμώμενο αντικείμενο της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes για την εκτέλεση των στοιχείων του εμπρόσθιου επιπέδου, διότι επιτρέπει την κλιμάκωσή του με τον ευκολότερο δυνατό τρόπο. Αναφορικά με την εκτέλεση του οπίσθιου επιπέδου, είχε αναφερθεί ότι χρησιμοποιούνται StatefulSet, αλλά τότε δεν είχαν δοθεί λεπτομέρειες για τον ακριβή τρόπο χρήσης τους.

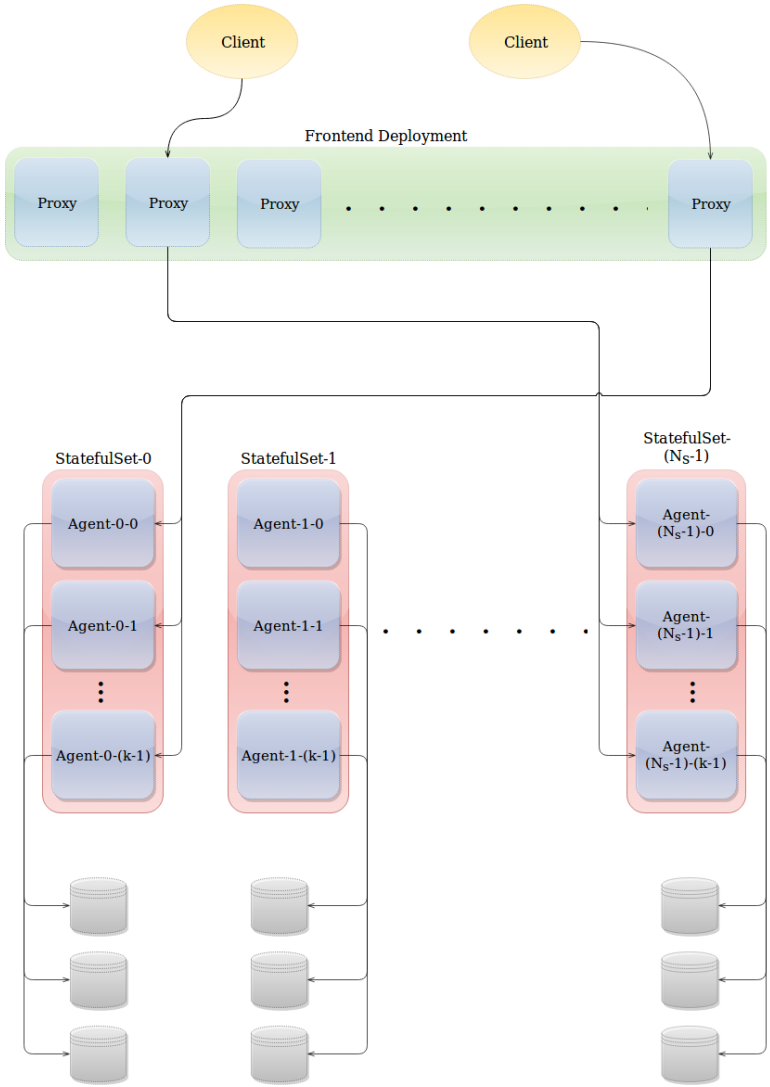
Στο σημείο αυτό, έχει έρθει η ώρα να εξετάσουμε αυτές τις λεπτομέρειες. Κάθε Πράκτορας στο MICAS είναι υπεύθυνος για τα δεδομένα ενός, και μόνο ενός, τεμαχίου. Όπως συζητήθηκε ακριβώς προηγουμένως, για τα δεδομένα κάθε τεμαχίου δημιουργούνται αντίγραφα, ουσιαστικά δημιουργώντας αντίγραφα για ολόκληρη την οντότητα που αντιστοιχεί σε ένα τεμάχιο· άρα, εν προκειμένω, τον Πράκτορα. Θέτοντάς το με άλλα λόγια, πολλαπλοί εκτελούμενοι Πράκτορες είναι υπεύθυνοι για το ίδιο ακριβώς σύνολο δεδομένων – τα δεδομένα ενός τεμαχίου – για κάθε τεμάχιο στο εύρος κλειδιών του συστήματος. Το ακριβές τους πλήθος είναι k, ίσο με τον παράγοντα αντιγραφής, το οποίο ρυθμίζεται, αλλά στατικά. Κάθε σύνολο Πρακτόρων (μεγέθους k) που είναι υπεύθυνοι για το ίδιο τεμάχιο, εκτελείται, χάρη στο Kubernetes, χρησιμοποιώντας Pods, τα οποία ελέγχονται από το ίδιο StatefulSet.

Σε καθένα από τα N_s τεμάχια, ανατίθεται ένας ακέραιος αριθμός από το εύρος $[0,\ N_s - 1]$, ως μοναδικό αναγνωριστικό, με στόχο την υλοποίηση του τεμαχισμού μέσω κατακερματισμού mod-N. Σε κάθε StatefulSet επίσης ανατίθεται ένας τέτοιος ακέραιος αριθμός ως αναγνωριστικό τεμαχίου, και συμπεριλαμβάνεται στο όνομά του. Αυτό, με τη σειρά του, επηρεάζει τόσο τη μοναδική διεύθυνση DNS του, η οποία είναι εσωτερική στη συστοιχία Kubernetes και παρέχεται μέσω του KubeDNS εκαταστήσιμου προσθέτου συστοιχίας, όσο και το μοναδικό αναγνωριστικό του για το Kubernetes (σε συνδυασμό με τον μηχανισμό των Namespaces του Kubernetes), επιτρέποντας με αυτό τον τρόπο την πρόσβαση σε κάθε Pod που ελέγχεται από το κάθε StatefulSet, οποιαδήποτε στιγμή, μέσω του μηχανισμού των Service (εν προκειμένω Headless) και των αντίστοιχων Endpoint. Από τη στιγμή που κάθε StatefulSet ελέγχει k Pods Πρακτόρων, παρατηρούμε ότι θα ήταν χρήσιμο να αναγνωρίζονται και τα ίδια τα Pods με μοναδικό τρόπο. Τα Pods, λοιπόν, εντός ενός StatefulSet, λαμβάνουν ένα μοναδικό ακέραιο αριθμο στο εύρος $[0,\ k-1]$ ως αναγνωριστικό, μέσω του ονόματός τους. Συνεπώς, εν τέλει, κάθε Pod Πράκτορα αναγνωρίζεται μέσω ενός μοναδικού ζεύγους αναγνωριστικών στο όνομά του, το οποίο είναι της μορφής "□□□□□□□□□□-i-

j", όπου το i δείχνει το αναγνωριστικό τεμαχίου και το j δείχνει το αναγνωριστικό αντιγράφου του Πράκτορα του i-οστού τεμαχίου. Το όνομα αυτής της μορφής, το οποίο ανατίθεται ντετερμινιστικά σε κάθε Pod Πράκτορα, σε συνδυασμό με το κατάλληλο Namespace του Kubernetes, μπορεί να χρησιμοποιηθεί τόσο από Διαμεσολαβητές όσο και από άλλους Πράκτορες, προκειμένου να επικοινωνήσουν μαζί του, είτε μέσω DNS, είτε μέσω του κατάλληλου Endpoint.

Το Σχήμα 7 απεικονίζει ένα σκαρίφημα της αρχιτεκτονικής του MICAS. Μπορεί κάποιος να διακρίνει ότι ένα οποιοδήποτε πλήθος Pods Διαμεσολαβητών μπορεί να εκτελεστεί στους κόμβους της συστοιχίας Kubernetes μέσω του "Frontend Deployment", καθώς και ένα στατικό πλήθος StatefulSet – ένα για καθένα εκ των N_s τεμαχίων – που το καθένα ελέγχει k Pods Πρακτόρων. Τα βέλη δείχνουν την πορεία των δεδομένων κατά τη διάρκεια εξυπηρέτησης ενός αιτήματος δημιουργίας ή ανάγνωσης κάποιου αντικειμένου.

Στο σημείο αυτό, είναι σημαντικό να αναφέρουμε ότι η δρομολόγηση των Pods Πρακτόρων στους φυσικούς κόμβους παίζει σημαντικό ρόλο. Πράκτορες που δρομολογούνται από το Kubernetes για εκτέλεση στους ίδιους κόμβους της συστοιχίας, είναι όλοι ευάλωτοι στις ίδιες αποτυχίες. Αφού όλοι οι k Πράκτορες ενός StatefulSet είναι υπεύθυνοι για το ίδιο τμήμα των δεδομένων που αποθηκεύονται συνολικά στο MICAS, Pods Πρακτόρων υπό το ίδιο StatefulSet δεν πρέπει να δρομολογούνται στον ίδιο φυσικό κόμβο. Διαφορετικά, αποτυχία ενός μόνο φυσικού κόμβου μπορεί να προκαλέσει πολλαπλά αντίγραφα των ίδιων αντικειμένων να είναι ταυτόχρονα μη διαθέσιμα. Ουσιαστικά, λοιπόν, πλήττεται η εγγύηση του συστήματος για ανοχή σε $k-1$ αποτυχίες, με την έννοια ότι $k-1$ Pods Πρακτόρων μπορεί να αποτύχουν ως αποτέλεσμα λιγότερων από $k-1$ αποτυχιών φυσικών κόμβων. Για να αντιμετωπίσουμε αυτό το πρόβλημα, χρησιμοποιούμε κατάλληλες επιλογές ρυθμίσεων του Kubernetes [180], ώστε μέσω Labels και Selectors να ορίσουμε κάποιους σχετικούς γενικούς κανόνες για την κατάλληλη δρομολόγηση και τοποθέτηση των Pods Πρακτόρων από το Kubernetes.

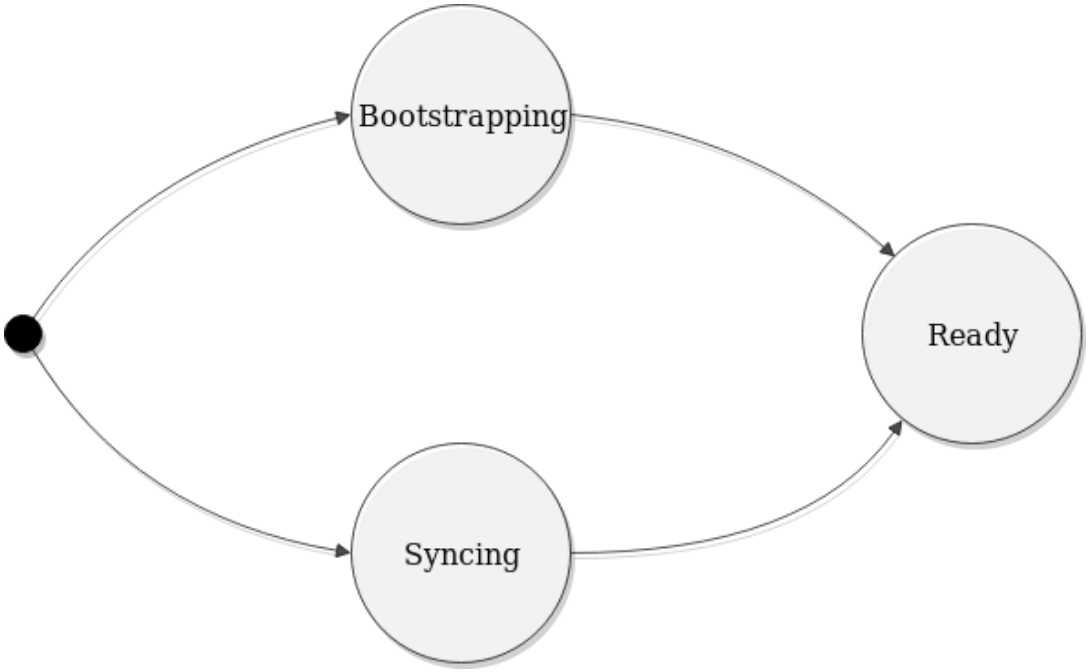


Σχήμα 7: *Η προτεινόμενη αρχιτεκτονική του* MICAS*, με ένα οποιοδήποτε πλήθος Διαμεσολαβητών, Πράκτορες πλήθους N_s, και με παράγοντα αντιγραφής k. Τα βέλη δείχνουν την πορεία των δεδομένων κατά τη διάρκεια εξυπηρέτησης ενός αιτήματος δημιουργίας ή ανάγνωσης αντικειμένου.*

Φάσεις Λειτουργίας

Πολύτιμος αρωγός στην παροχή κλιμακωσιμότητας και αυτοΐασης του MICAS, όπως και του RICAS που θα εξετάσουμε αργότερα, είναι το Kubernetes, το οποίο, όπως διαπιστώνουμε εκ των έως τώρα συζητηθέντων, βρίσκεται στον πυρήνα της σχεδίασής μας. Ο ρόλος του δεν περιορίζεται μόνο στη στατική δόμηση του συστήματος, όπως έχουμε εξετάσει μέχρι στιγμής, αλλά εκτείνεται σε όλη τη διάρκεια της ζωής κάθε επιμέρους συνιστώσας του.

Αφού ξεκινήσει τη λειτουργία του και ολοκληρώσει τις τοπικές του αρχικοποιήσεις,



Σχήμα 8: *Διάγραμμα καταστάσεων ενός Πράκτορα* `MICAS`. *Ξεκινώντας την εκτέλεσή του, ένας Πράκτορας* `MICAS` *μπορεί να βρεθεί σε μία εκ των φάσεων Εκκίνησης ή Συγχρονισμού. Σε κάθε περίπτωση, καταλήγει στη φάση Ετοιμότητας.*

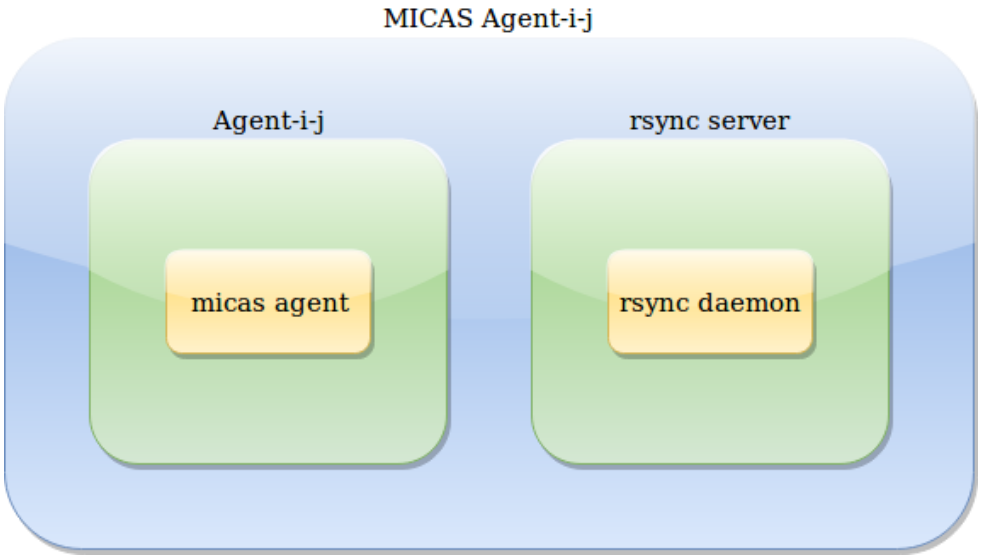
ένας `MICAS` Πράκτορας μπορεί να βρεθεί είτε στη φάση Εκκίνησης ("Bootstrapping phase"), είτε στη φάση Συγχρονισμού ("Syncing phase"). Στην πρώτη περίπτωση, το `MICAS` πρέπει να έχει μόλις "παραταχθεί" στη συστοιχία. Κάθε Πράκτορας περιμένει όλους τους υπόλοιπους να ξεκινήσουν τη λειτουργία τους και να ολοκληρώσουν τις δικές τους αρχικοποιήσεις, ούτως ώστε όλοι μαζί να μεταβούν στην επόμενη φάση, τη φάση Ετοιμότητας ("Ready phase"), όπως φαίνεται στο Σχήμα 8. Εδώ, είναι προφανής η υποβόσκουσα παρουσία της έννοιας της κατανεμημένης συμφωνίας, η οποία επιτυγχάνεται χάρη στην παροχή "αισιόδοξου ελέγχου συγχρονικότητας"[64] μέσω των αντικειμένων της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes [183].

Η φάση Ετοιμότητας σημαίνει απλώς ότι ο Πράκτορας είναι έτοιμος να δεχθεί και να εξυπηρετήσει αιτήματα πελατών που καταφθάνουν από τους Διαμεσολαβητές.

Στην περίπτωση που ένας Πράκτορας ο οποίος έχει μόλις αρχίσει τη λειτουργία του συνειδητοποιήσει ότι δεν βρίσκεται στη φάση Εκκίνησης, δηλαδή ότι δεν πρόκειται για την έναρξη λειτουργίας του `MICAS`, σημαίνει ότι ο ίδιος επανέρχεται από σφάλμα ή αποτυχία κόμβου. Στην περίπτωση αυτή, ο Πράκτορας μεταβαίνει στη φάση Συγχρο-

[64] Ο ελληνικός όρος "αισιόδοξος έλεγχος συγχρονικότητας" εκφράζει την έννοια του ευρέως γνωστού αγγλικού όρου "optimistic concurrency control".

νισμού, κατά την οποία, χρησιμοποιώντας, όπως προαναφέρθηκε, το rsync, πραγματοποιείται ο συγχρονισμός δεδομένων εκκίνησης. Για να επιτευχθεί αυτό, ο Πράκτορας επικοινωνεί, μέσω του rsync, με τους rsync δαίμονες των υπόλοιπων Πρακτόρων που διατηρούν αποθηκευμένα αντίγραφα των αντικειμένων του τεμαχίου για το οποίο είναι και ο ίδιος υπεύθυνος, ενώ ταυτόχρονα εξυπηρετεί και νέα αιτήματα πελατών που του προωθούνται μέσω των Διαμεσολαβητών. Η διάταξη των περιεκτών και των διεργασιών του Pod του Πράκτορα `MICAS` είναι αυτή που είχε περιγραφθεί νωρίτερα· είναι απλή, όπως φαίνεται στο Σχήμα 9. Το Pod του `MICAS` Πράκτορα αποτελείται από δύο περιέκτες, έναν για τη διεργασία που εκτελεί τη λογική του Πράκτορα, και έναν rsync δαίμονα που εξυπηρετεί εισερχόμενα αιτήματα συγχρονισμού δεδομένων από τους υπόλοιπους Πράκτορες του ίδιου `StatefulSet` κατά τη φάση Συγχρονισμού τους.



Σχήμα 9: *Το Pod του* `MICAS` *Πράκτορα αποτελείται από δύο περιέκτες, έναν για τη διεργασία που εκτελεί τη λογική του Πράκτορα, και έναν rsync δαίμονα που εξυπηρετεί εισερχόμενα αιτήματα συγχρονισμού δεδομένων από τους υπόλοιπους Πράκτορες του ίδιου* `StatefulSet` *κατά τη φάση Συγχρονισμού τους.*

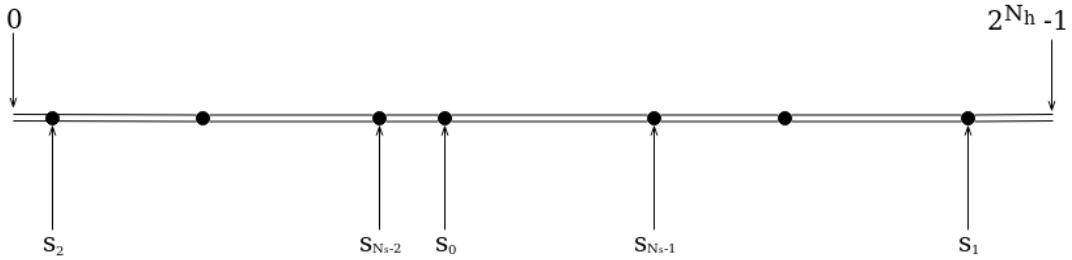
RICAS

Το `RICAS`, το οποίο είναι ακρωνύμιο για το "Ring-based Immutable Content-Addressable Storage", είναι το αποτέλεσμα της δεύτερης μας προσέγγισης στη σχεδίαση του κατανεμημένου συστήματος αποθήκευσης, και το δεύτερο από τα δύο συστήματα που

παρουσιάζονται στα πλαίσια της παρούσας εργασίας.

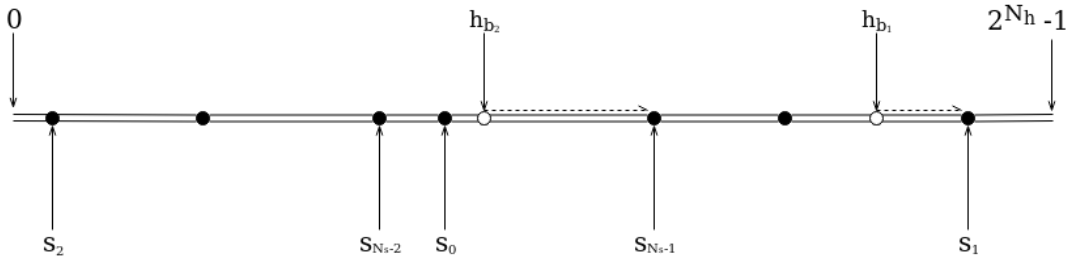
Τεμαχισμός με Συνεπή Κατακερματισμό

Η λογική του τεμαχισμού στο RICAS βασίζεται στον αλγόριθμο "συνεπούς κατακερματισμού" ("consistent hashing") [21]. Σύμφωνα με αυτόν, η σύνοψη κατακερματισμού N_h δυφίων του μοναδικού αναγνωριστικού κάθε Πράκτορα RICAS αρχικά "τοποθετείται" πάνω στον άξονα των ακέραιων αριθμών στο εύρος $[0,\ N_h - 1]$, όπως φαίνεται στο Σχήμα 10.



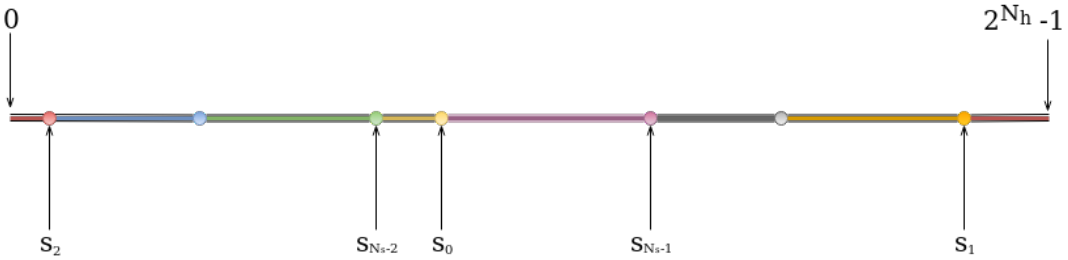
Σχήμα 10: *Οι συνόψεις κατακερματισμού N_s Πρακτόρων πάνω στον άξονα ακεραίων στο εύρος $[0,\ N_h - 1]$, όπου το S_i αναπαριστά τον i-οστό Πράκτορα με $i \in \{0,\ \ldots,\ N_s - 1\}$.*

Δεδομένων ενός μεγάλου δυαδικού αντικειμένου, b, και της σύνοψης κατακερματισμού του $h_b = h(b)$, αναζητούμε επί του άξονα, δεξιά του h_b μέχρι να βρεθεί ένας ακέραιος S_j ο οποίος αναπαριστά τη σύνοψη κατακερματισμού ενός Πράκτορα. Τότε, το μεγάλο δυαδικό αντικείμενο b ανατίθεται στον j-οστό Πράκτορα που αντιστοιχεί στη σύνοψη κατακερματισμού S_j.Ένα τέτοιο παράδειγμα φαίνεται στο Σχήμα 11, όπου τα αντικείμενα με κλειδιά $h_{b_1} = h(b_1)$ και $h_{b_2} = h(b_2)$ ανατίθενται στους Πράκτορες S_1 και S_{N_s-1}, αντιστοίχως, βάσει αυτής της λογικής.



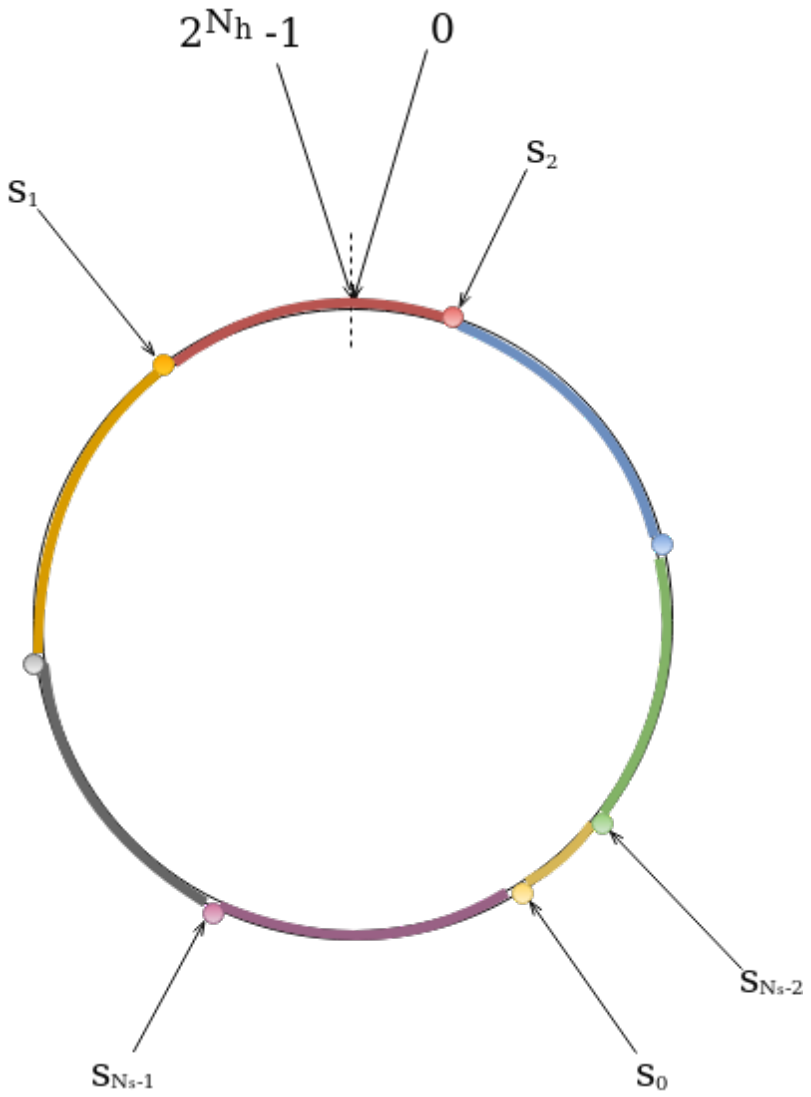
Σχήμα 11: *Ο άξονας του Σχήματος 10, επισημειωμένος ώστε να περιλαμβάνει τα κλειδιά των αντικειμένων $h_{b_1} = h(b_1)$ και $h_{b_2} = h(b_2)$. Παρατηρούμε τη λογική ανάθεσης του h_{b_1} στον Πράκτορα S_1 και του h_{b_2} στον Πράκτορα S_{N_s-1}.*

Με άλλα λόγια, κάθε Πράκτορας που τοποθετείται στον άξονα αναπαριστά ένα τεμάχιο του εύρους κλειδιών, το οποίο περιλαμβάνει κάθε αντικείμενο του οποίου το κλειδί, ως σύνοψη κατακερματισμού και συνεπώς ακέραιος αριθμός, εμπίπτει στο εύρος των ακεραίων μεταξύ της σύνοψης κατακερματισμού του ιδίου και του προηγούμενού του επί του άξονα Πράκτορα. Στο Σχήμα 12, απεικονίζεται ο άξονας του Σχήματος 11, χρησιμοποιώντας από ένα μοναδικό χρώμα για κάθε Πράκτορα και το αντίστοιχο τεμάχιό του.



Σχήμα 12: *Ο άξονας του Σχήματος 11, χρησιμοποιώντας από ένα μοναδικό χρώμα για κάθε Πράκτορα και το αντίστοιχο τεμάχιό του. Βλέπουμε με ποιον τρόπο γίνεται η ανάθεση των περιπτώσεων των στοιχείων του ακραίου δεξιού επί του άξονα τεμαχίου στον ακραίο αριστερό επί του άξονα Πράκτορα.*

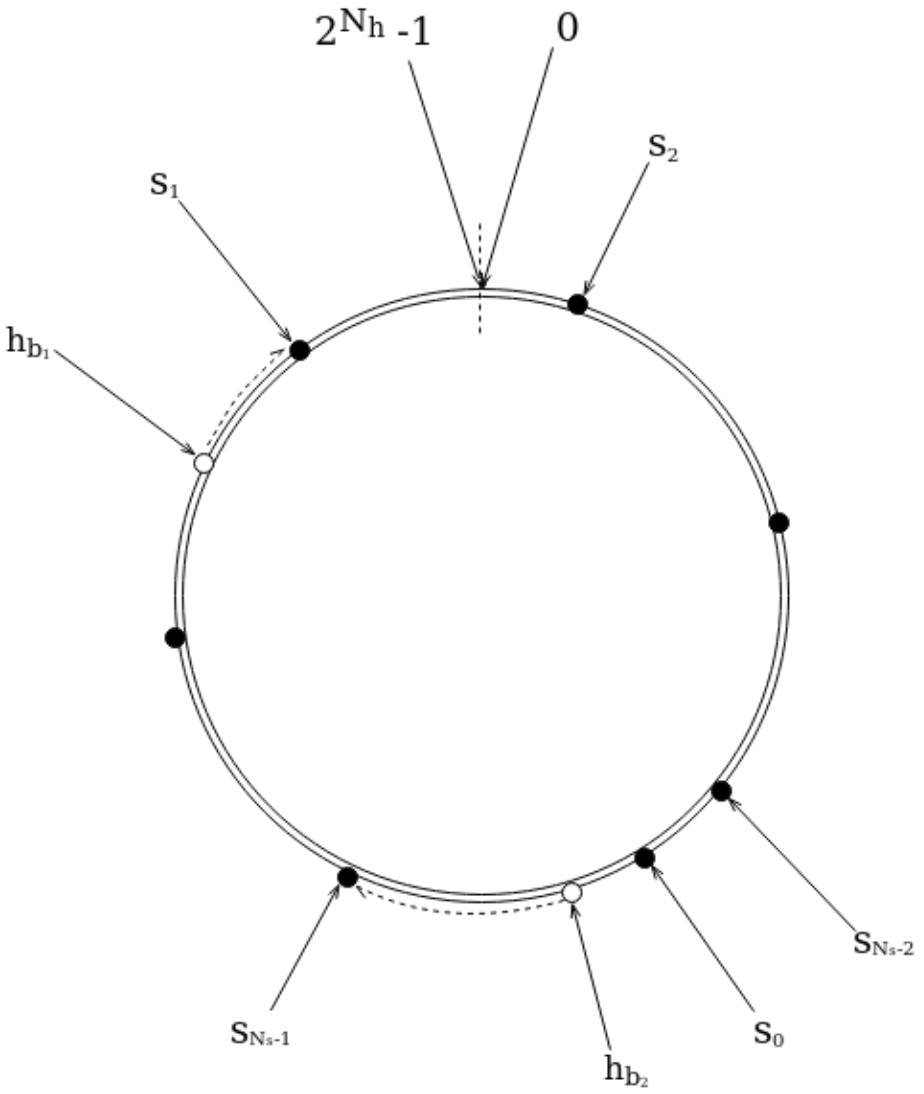
Είναι ολοφάνερο ότι αντικείμενα στο ακραίο δεξιό επί του άξονα τεμάχιο δεν έχουν κανέναν Πράκτορα προς τα δεξιά τους στον οποίον να ανατεθούν. Σε τέτοιες περιπτώσεις, ανατίθενται στον ακραίο αριστερό επί του άξονα Πράκτορα, όπως φαίνεται στο Σχήμα 12. Αυτό έχει ως αποτέλεσμα τη δυνατότητα θεώρησης του άξονα αυτού ως δακτύλιο, το οποίο με τη σειρά του δημιουργεί την ευρέως διαδεδομένη ορολογία "δακτύλιος συνεπούς κατακερματισμού" ("consistent hashing ring"), από το οποίο είναι εμπνευσμένο και το όνομα του `RICAS`. Για την οπτικοποίηση αυτής της συλλογιστικής, το Σχήμα 13 αναπαριστά τον άξονα του Σχήματος 12 ως δακτύλιο.



Σχήμα 13: *Το εύρος κλειδιών του άξονα του Σχήματος 12 μπορεί να "τοποθετηθεί" σε έναν κύκλο αντί ενός ευθύγραμμου τμήματος, και έτσι να αναπαρασταθεί ως ένας δακτύλιος, ο "δακτύλιος συνεπούς κατακερματισμού".*

Επί του δακτυλίου συνεπούς κατακερματισμού, η διαδικασία ανάθεσης αντικειμένων σε Πράκτορες γίνεται με παρόμοιο τρόπο όπως και στην περίπτωση του άξονα, με την επισήμανση ότι η αναζήτηση Πράκτορα πλέον γίνεται κινούμενοι δεξιόστροφα, δηλαδή κατά την ωρολογιακή φορά. Το ίδιο παράδειγμα που παρουσιάστηκε προηγουμένως στο Σχήμα 11 επί του άξονα, παρουσιάζεται ξανά στο Σχήμα 14 επί του δακτυλίου, ώστε να γίνει καλύτερα κατανοητή η μικρή αλλαγή στη συλλογιστική.

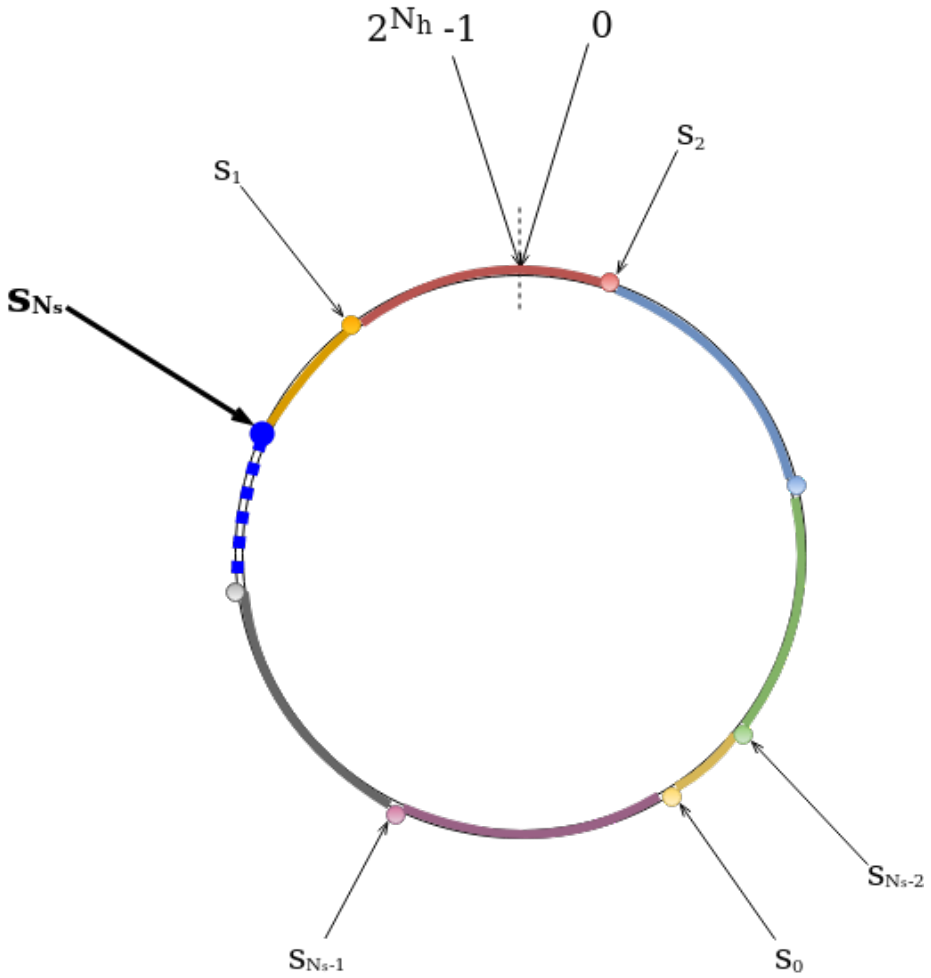
Το βασικό χαρακτηριστικό του συνεπούς κατακερματισμού ως μέθοδος τεμαχισμού είναι η ελαστικότητα που παρέχει, αφού επιτρέπει εύκολα διαχειρίσιμο και αποδοτικό επανατεμαχισμό του εύρους κλειδιών και των αποθηκευμένων δεδομένων του συστή-



Σχήμα 14: *Η διαδικασία τεμαχισμού που παρουσιάστηκε στο Σχήμα 11 επί του άξονα, αναπαριστώμενη και επί του δακτυλίου συνεπούς κατακερματισμού.*

ματος. Η προσθήκη ενός νέου Πράκτορα στη συστοιχία RICAS σημαίνει την τοποθέτηση της σύνοψης κατακερματισμού του στον δακτύλιο κατακερματισμού, και άρα τη διαμόρφωση ενός νέου τεμαχίου, το οποίο μέχρι πρότινος ήταν τμήμα κάποιου από τα προϋπάρχοντα τεμάχια. Εντελώς ανάλογα, η αφαίρεση ενός Πράκτορα σημαίνει την αφαίρεση της σύνοψης κατακερματισμού του από τον δακτύλιο κατακερματισμού, και άρα την συγχώνευση του αντίστοιχου τεμαχίου του με το πλησιέστερό του κατά την ωρολογιακή φορά. Η προσθήκη (ή αφαίρεση) ενός Πράκτορα από τον δακτύλιο κατακερματισμού, μαζί με το αντίστοιχο τεμάχιο, απεικονίζεται στο Σχήμα 15.

Αποδεικνύεται[21] ότι ο αλγόριθμος συνεπούς κατακερματισμού έχει διάφορες σημαντικές και χρήσιμες ιδιότητες, εκ των οποίων ξεχωρίζουμε και αναφέρουμε τις δύο:



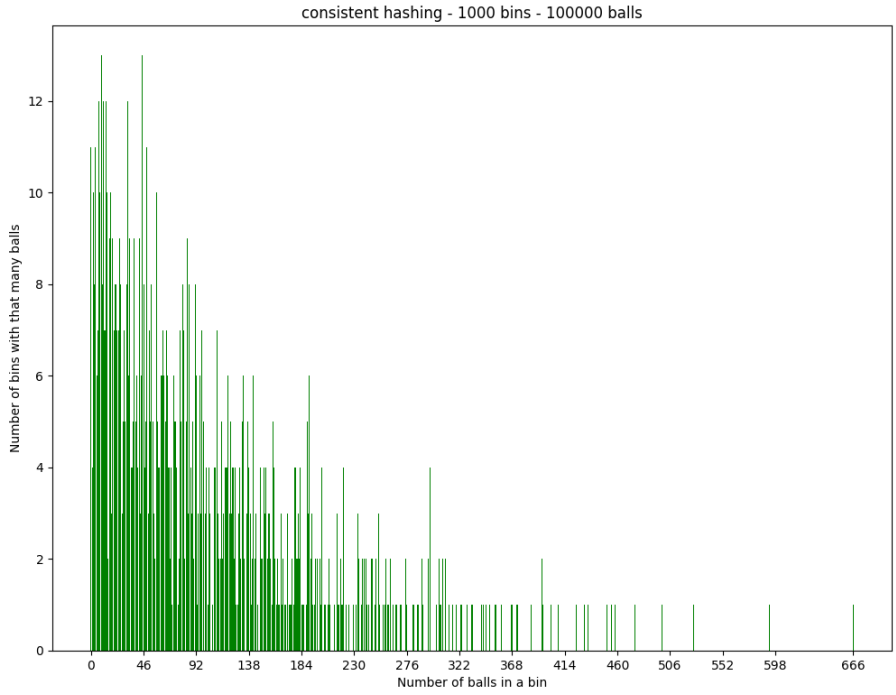
Σχήμα 15: *Η προσθήκη ή αφαίρεση ενός Πράκτορα* RICAS*, του N_s, και του τεμαχίου που του αντιστοιχεί, από τον δακτύλιο συνεπούς κατακερματισμού του συστήματος.*

- το ποσοστό των αντικειμένων που απαιτείται να μετακινηθούν κατά την προσθήκη ή αφαίρεση Πρακτόρων από τον δακτύλιο είναι το ελάχιστο δυνατό για τη διατήρηση εξισορροπημένου φόρτου μεταξύ των Πρακτόρων·

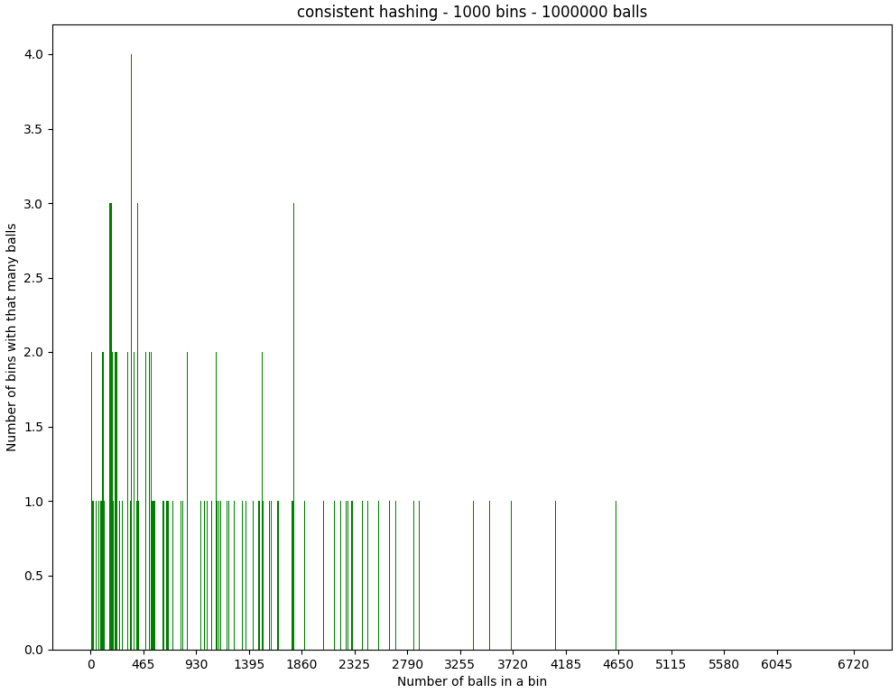
- κατά την προσθήκη ή αφαίρεση Πρακτόρων, η ροή των μεταφερόμενων αντικειμένων έχει φορά πάντα από Πράκτορες παρόντες στην παλαιά κατάσταση του δακτυλίου προς Πράκτορες παρόντες στη νέα κατάσταση του δακτυλίου. Δηλαδή, κατά την προσθήκη Πρακτόρων, δεδομένα μεταφέρονται μόνο από κάποιους από τους προϋπάρχοντες στον δακτύλιο Πράκτορες προς τους νεοεισεχθέντες Πράκτορες, ενώ κατά την αφαίρεση Πρακτόρων, δεδομένα μεταφέρονται μόνο από τους Πράκτορες που πρόκειται να αφαιρεθούν προς κάποιους από τους Πράκτορες που πρόκειται να παραμείνουν στον δακτύλιο.

Το βασικό μειονέκτημα της τεχνική τεμαχισμού μέσω του αλγορίθμου συνεπούς κατακερματισμού που χρησιμοποιείται στο RICAS σε σχέση με τον τεμαχισμό μέσω κατακερματισμού mod-N που χρησιμοποιείται στο MICAS, είναι σχετικό με την εξισορρόπηση φόρτου. Όπως είδαμε και στα προηγούμενα, το πρόβλημα μπορεί να μοντελοποιηθεί μέσω του κλασικού προβλήματος της Θεωρίας Πιθανοτήτων με τις "μπάλες" που εναποτίθενται σε "κουβάδες". Όπως και στην περίπτωση του MICAS και του κατακερματισμού mod-N, έτσι και για το RICAS και τον αλγόριθμο συνεπούς κατακερματισμού εκτελέσαμε μία σειρά προσομειώσεων Monte Carlo, και κατασκευάσαμε γραφικές παραστάσεις για να αποφανθούμε περί της αποδοτικότητας της διανομής των μπαλών στους κουβάδες όσον αφορά την εξισορρόπηση του φορτίου μεταξύ τους. Στις προσομειώσεις, το πλήθος των κουβάδων παραμένει σταθερό και ίσο με $N_s = 1000$ μπάλες, ενώ το πλήθος των διανεμητέων μπαλών, n, μεταβάλλεται. Στις γραφικές παραστάσεις των Σχημάτων 16 και 17, ο οριζόντιος άξονας δείχνει τις τιμές του πλήθους μπαλών ανά κουβά, ενώ ο κατακόρυφος άξονας δείχνει το πλήθος των κουβάδων που έχει όσες μπάλες δείχνει ο οριζόντιος.

Ιδανικά, σε όλες τις προσωμειώσεις, το πλήθος μπαλών ανά κουβά, το οποίο μοντελοποιεί το πλήθος των αντικειμένων ανά τεμάχιο, θα έπρεπε να είναι κατανεμημένο γύρω από το $\frac{n}{N_s}$. Παρ'ολ'αυτά, προφανώς κάτι τέτοιο δεν ισχύει. Είναι ολοφάνερο ότι σε κάποιους κουβάδες έχουν ανατεθεί λιγότερες από το $\frac{n}{N_s}$ των μπαλών, της τάξεως του $0.46\frac{n}{N_s}$ ή και λιγότερο, ενώ σε άλλους κουβάδες έχουν ανατεθεί αρκετές περισσότερες μπάλες, ακόμα και της τάξεως του $6.7\frac{n}{N_s}$ σε μερικές περιπτώσεις.

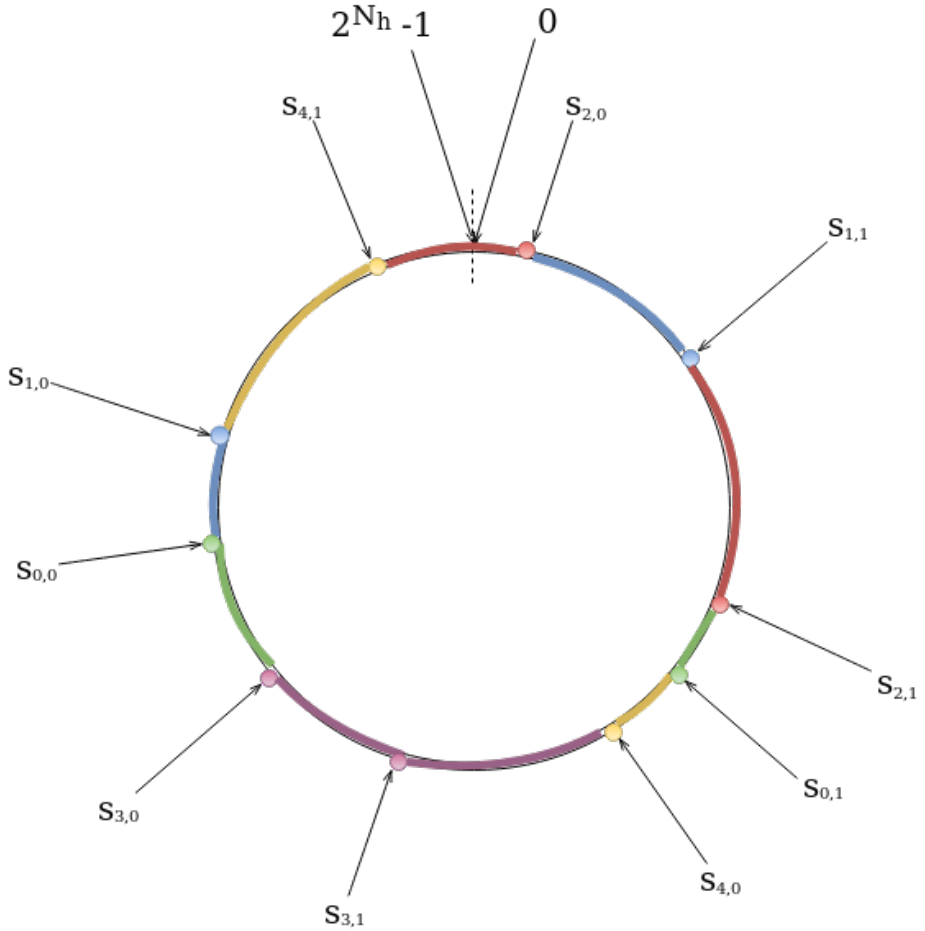


Σχήμα 16: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες και* $n = 10^5$ *μπάλες.* $\frac{n}{N_s} = 100$ *μπάλες ανά κουβά.*



Σχήμα 17: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες και* $n = 10^6$ *μπάλες.* $\frac{n}{N_s} = 1000$ *μπάλες ανά κουβά.*

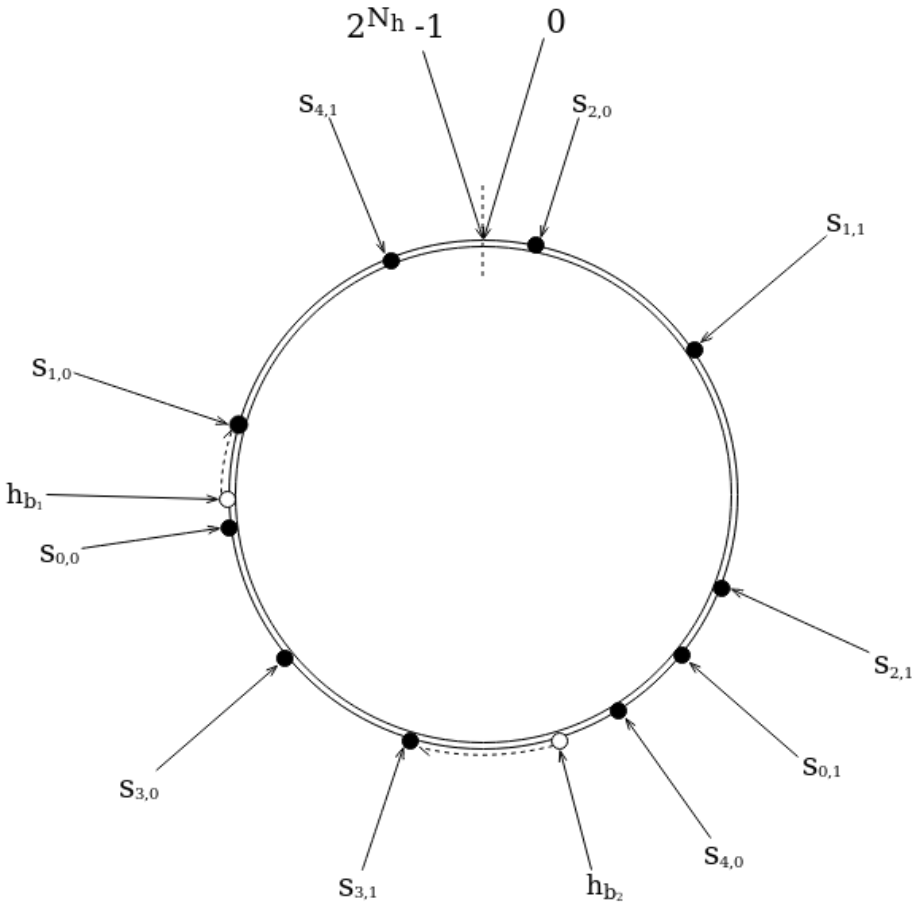
Αισίως, υπάρχει ένας γνωστός τρόπος για τη μείωση της μεγάλης διασποράς που παρατηρείται στις παραπάνω γραφικές παραστάσεις. Αντί να "τοποθετούμε" τη σύνοψη κατακερματισμού κάθε Πράκτορα από μία φορά πάνω στον δακτύλιο κατακερματισμού, υπολογίζουμε V συνόψεις κατακερματισμού για τον καθένα, χρησιμοποιώντας από μία ακολουθία V μοναδικών αναγνωριστικών προερχόμενα από το αρχικό μοναδικό αναγνωριστικό του. Αυτή η τεχνική λέγεται "εικοινικοί κόμβοι", και με αυτόν τον τρόπο, N_s Πράκτορες διαμερίζουν τον δακτύλιο σε $V \cdot N_s$ τμήματα, με αναμενόμενο μέσο φόρτο περίπου $\frac{1}{V \cdot N_s}$ το καθένα. Ένα παράδειγμα τοποθέτησης συνόψεων κατακερματισμού $N_s = 5$ Πρακτόρων με $V = 2$ εικονικούς κόμβους έκαστος, δημιουργώντας συνεπώς 10 τμήματα συνολικά, πάνω στον δακτύλιο κατακερματισμού, φαίνεται στο Σχήμα 18.



Σχήμα 18: *Ένας δακτύλιος κατακερματισμού αποτελούμενος από $N_s = 5$ Πράκτορες με $V = 2$ εικονικούς κόμβους ο καθένας. Κάθε τμήμα του δακτυλίου εκπροσωπείται από την αντίστοιχη σύνοψη κατακερματισμού του εικονικού κόμβου, $S_{i,j}$, όπου $i \in \{0, \ldots, N_s - 1\}$ και $j \in \{0, \ldots, V - 1\}$.*

Η λογική της ανάθεσης των αντικειμένων σε Πράκτορες παραμένει κατά τα άλλα η

ίδια, με τη διαφορά ότι τώρα λαμβάνεται υπ'όψιν και η παρουσία των εικονικών κόμβων. Έτσι, στο Σχήμα 19 βλέπουμε με ποιον τρόπο τα αντικείμενα b_1 και b_2, με συνόψεις κατακερματισμού $h_{b_1} = h(b_1)$ και $h_{b_2} = h(b_2)$, ανατίθενται στους κόμβους S_1 και S_3, αντιστοίχως, αφού η ωρολογιακής φοράς διάσχιση του δακτυλίου οδηγεί στους εικονικούς κόμβους $S_{1,0}$ και $S_{3,1}$.

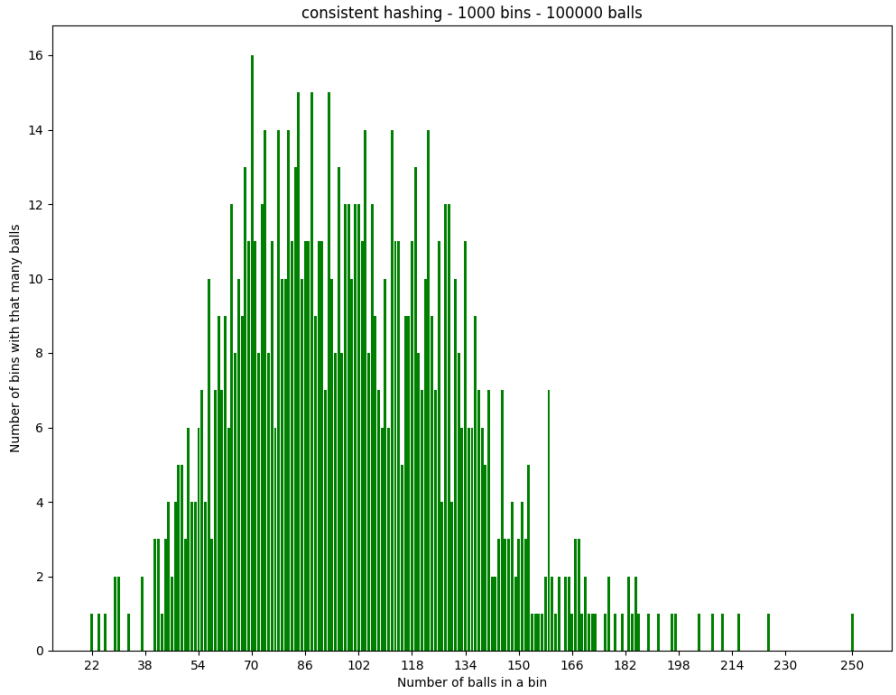


Σχήμα 19: *Η διαδικασία τεμαχισμού παρουσία εικονικών κόμβων.*

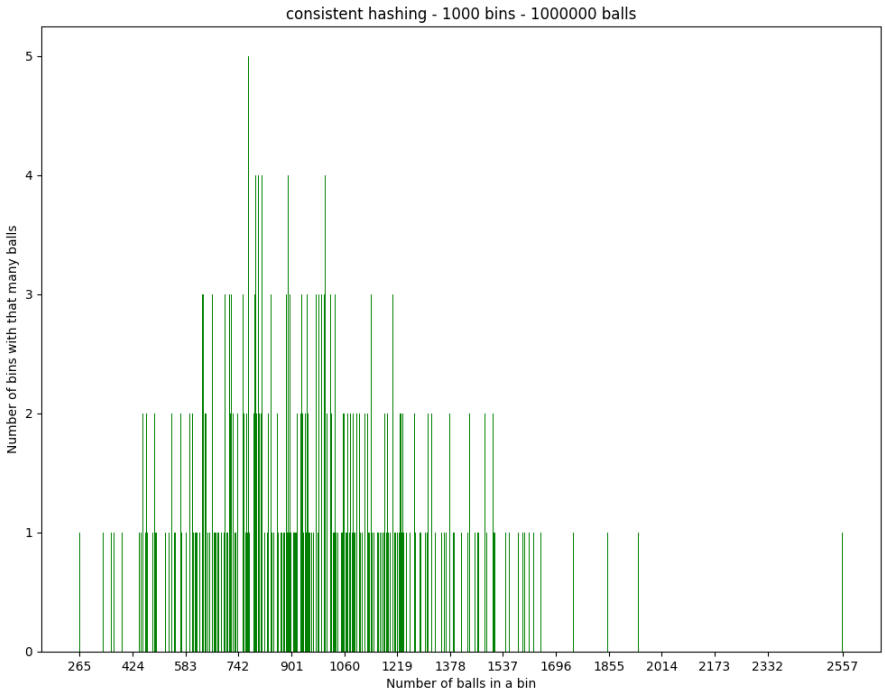
Η κλιμάκωση του συστήματος γίνεται με τον ίδιο τρόπο από την άποψη του δακτυλίου συνεπούς κατακερματισμού, με τη διαφορά ότι αυτή τη φορά κάθε Πράκτορας έχει πολλαπλούς γειτονικούς Πράκτορες (από έναν για καθέναν από τους V εικονικούς του κόμβους) από ή προς τους οποίους πρέπει να μεταφέρει τα δεδομένα του.

Σύμφωνα με τη μοντελοποίηση του προβλήματος βάσει του προβλήματος μπαλών και κουβάδων, επανεκτελούμε τις Monte Carlo προσομειώσεις για να διαπιστώσουμε κατά πόσον πράγματι βελτιώθηκε η εξισορρόπηση φόρτου στο `RICAS` με την εφαρμογή της τεχνικής των εικονικών κόμβων. Στα Σχήματα 20, 21, 22, 23, 24, 25 το πλήθος

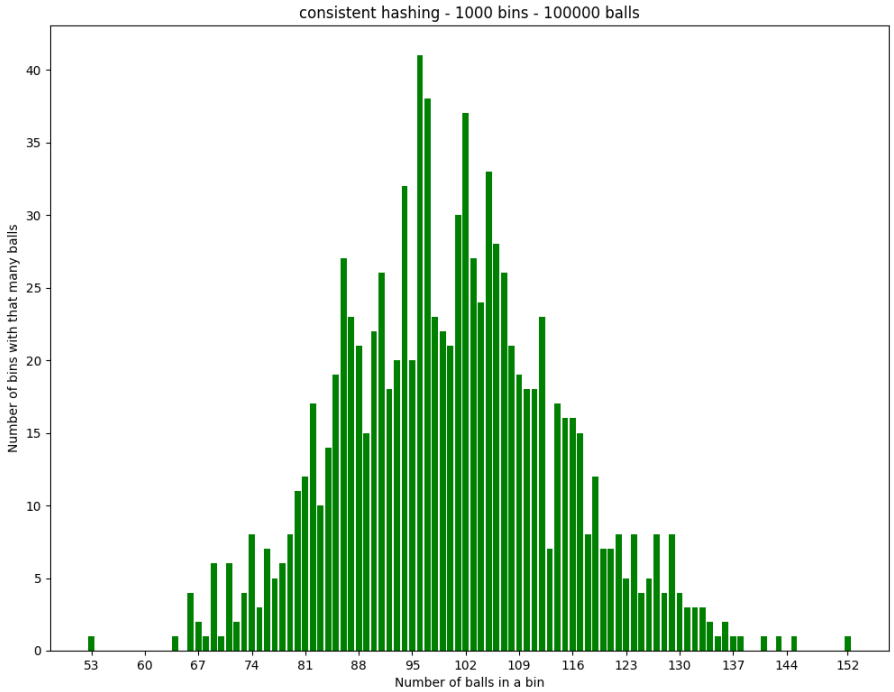
των κουβάδων έχει μείνει σταθερό και ίσο με $N_s = 1000$, ενώ οι τιμές των V και n μεταβάλλονται για να εξετάσουμε πώς διαφορετικές τιμές του πλήθους των εικονικών κόμβων ανά Πράκτορα επηρεάζουν την εξισορρόπηση φόρτου.



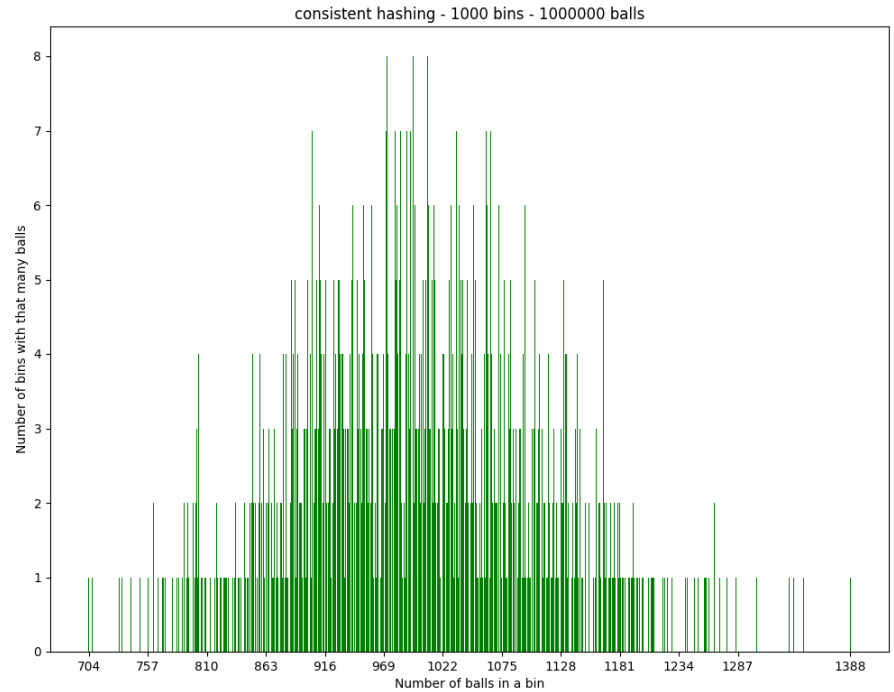
Σχήμα 20: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για $N_s = 1000$ κουβάδες, $V = 10$ εικονικούς κάδους και $n = 10^5$ μπάλες. $\frac{n}{N_s} = 100$ μπάλες ανά κουβά.*



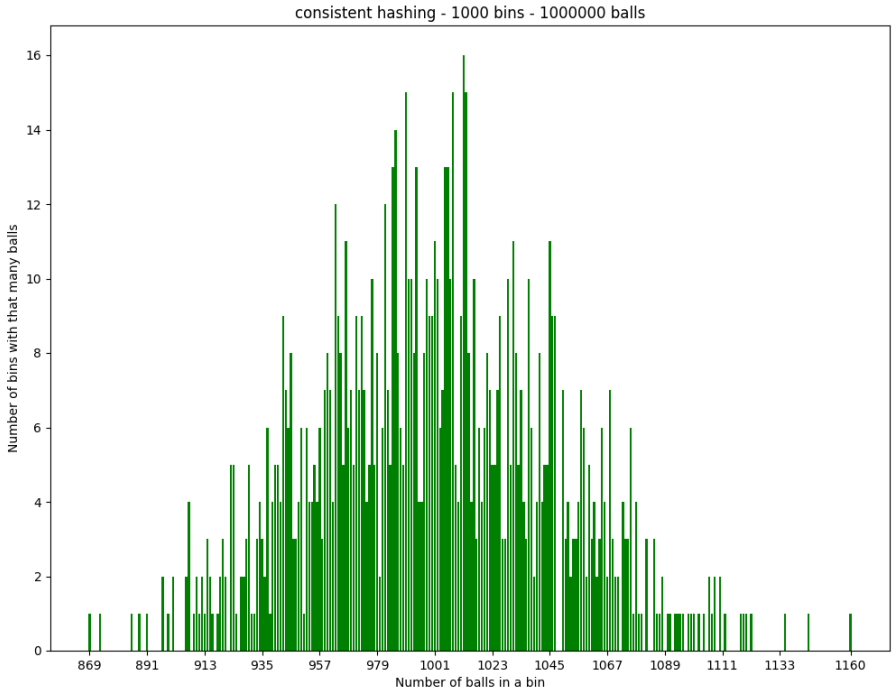
Σχήμα 21: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για $N_s = 1000$ κουβάδες, $V = 10$ εικονικούς κάδους και $n = 10^6$ μπάλες. $\frac{n}{N_s} = 1000$ μπάλες ανά κουβά.*



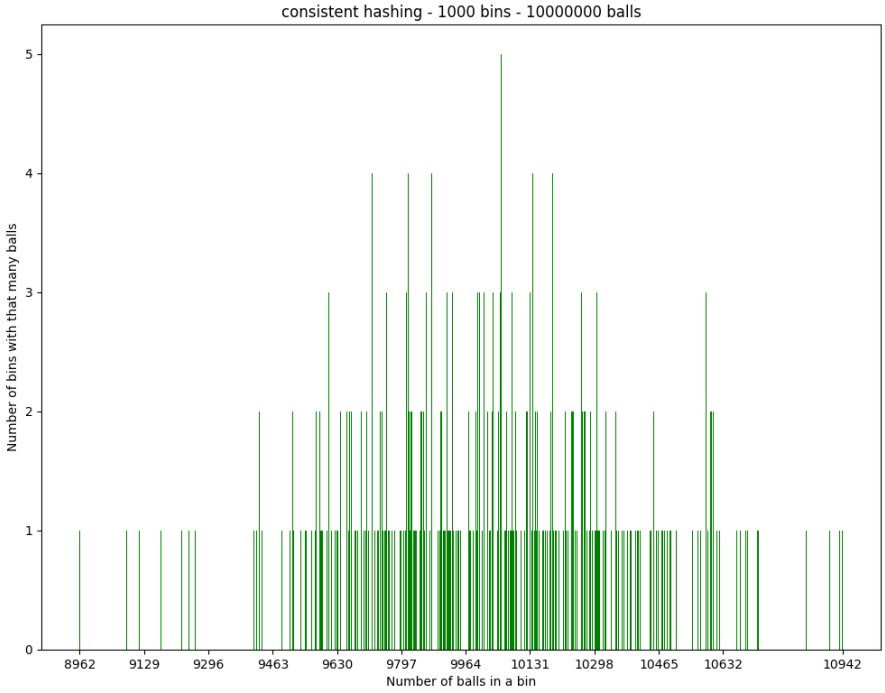
Σχήμα 22: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες,* $V = 100$ *εικονικούς κάδους και* $n = 10^5$ *μπάλες.* $\frac{n}{N_s} = 100$ *μπάλες ανά κουβά.*



Σχήμα 23: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες,* $V = 100$ *εικονικούς κάδους και* $n = 10^6$ *μπάλες.* $\frac{n}{N_s} = 1000$ *μπάλες ανά κουβά.*



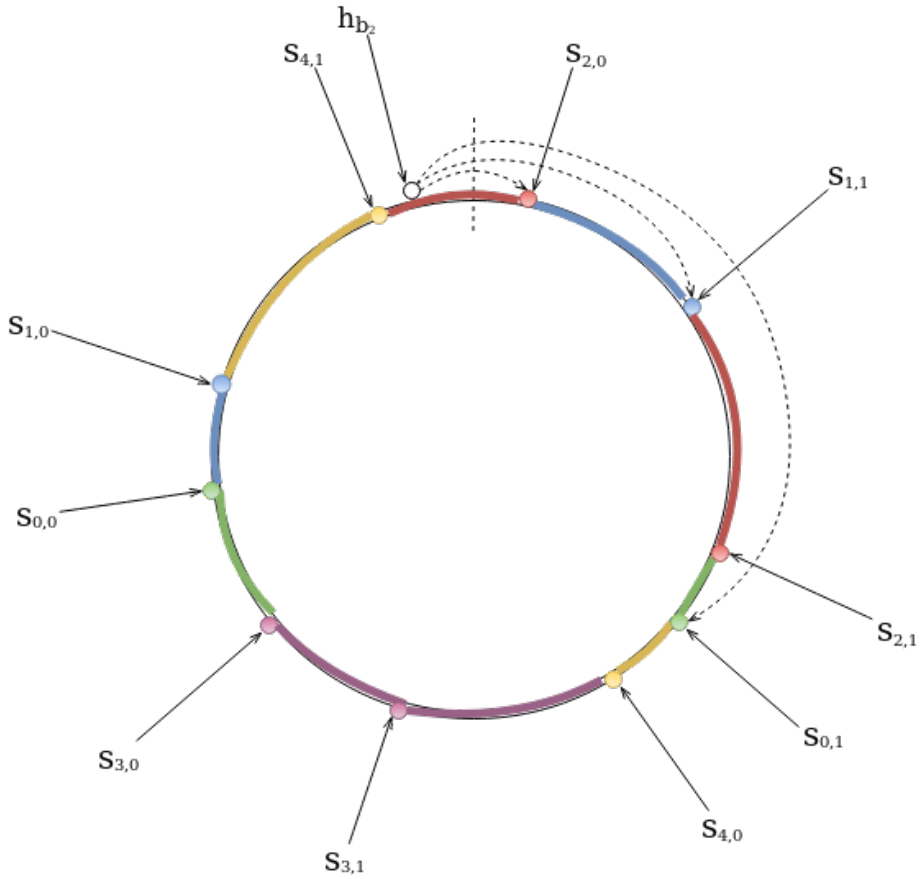
Σχήμα 24: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες,* $V = 1000$ *εικονικούς κάδους και* $n = 10^6$ *μπάλες.* $\frac{n}{N_s} = 1000$ *μπάλες ανά κουβά.*



Σχήμα 25: *Κατανομή μπαλών ανά κουβά χρησιμοποιώντας συνεπή κατακερματισμό για* $N_s = 1000$ *κουβάδες,* $V = 1000$ *εικονικούς κάδους και* $n = 10^7$ *μπάλες.* $\frac{n}{N_s} = 10000$ *μπάλες ανά κουβά.*

Αυτοματοποιημένη Δημιουργία Αντιγράφων

Η αυτοματοποιημένη δημιουργία αντιγράφων των αντικειμένων προς αποθήκευση μπορεί να πραγματοποιηθεί με πολλούς διαφορετικούς τρόπους. Εκείνος που τελικά επιλέχθηκε για τη σχεδίαση και την υλοποίησή μας βασίζεται σε μια μικρή επέκταση του αλγορίθμου ανάθεσης των αντικειμένων σε Πράκτορες. Η αναζήτηση επί του δακτυλίου για την εύρεση του κατάλληλου κάθε φορά Πράκτορα παραμένει η ίδια, με την διαφορά ότι από τη στιγμή που αυτός βρεθεί, το αντικείμενο ανατίθεται στους k διαδοχικούς Πράκτορες (και όχι εικονικούς κόμβους) επί του δακτυλίου στο σημείο εκείνο, όπου k είναι ο παράγοντας αντιγραφής.



Σχήμα 26: *Παράδειγμα της διαδικασίας αυτόματης δημιουργίας αντιγράφων των προς αποθήκευση αντικειμένων.*

Στο Σχήμα 26, θεωρώντας παράγοντα αντιγραφής $k = 3$ παρατηρούμε με ποιον τρόπο το μεγάλο δυαδικό αντικείμενο b_2, με σύνοψη κατακερματισμού $h_{b_2} = h(b_2)$, ανατίθεται μέσω πολλαπλών αντιγράφων σε Πράκτορες επί του δακτυλίου κατακερματισμού.

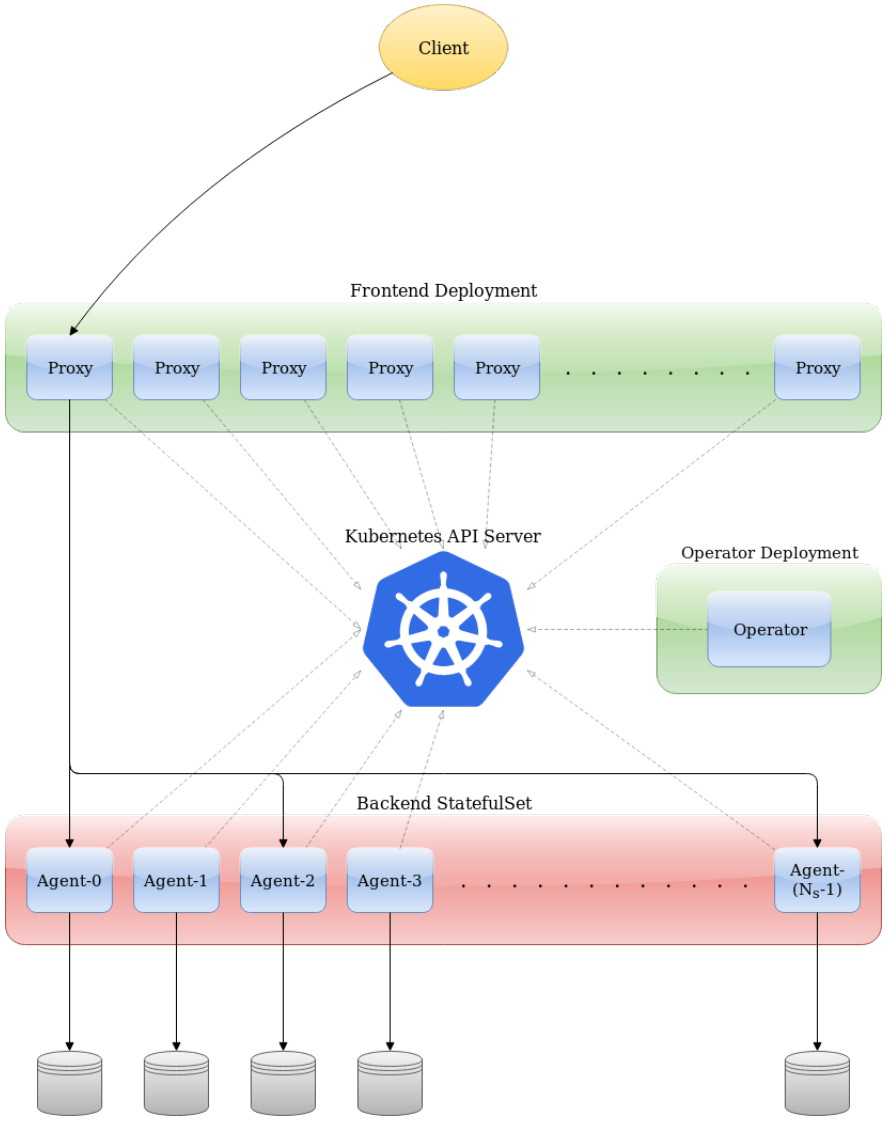
Αρχιτεκτονική

Όπως και το MICAS, έτσι και το παρόν σύστημα αποτελείται από ένα εμπρόσθιο επίπεδο Διαμεσολαβητών και ένα οπίσθιο επίπεδο Πρακτόρων. Σε αντίθεση όμως με το πρώτο, η αρχιτεκτονική του RICAS περιλαμβάνει ένα ακόμα στοιχείο, ονόματι Operator, το οποίο συμβάλλει στη διαχείριση ενός αντικειμένου της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes που το RICAS ορίζει κάνοντας χρήση του μηχανισμού CustomResourceDefinition που παρέχει το Kubernetes.

Τα υψηλότερου επιπέδου αντικείμενα της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes που χρησιμοποιούνται, παραμένουν ουσιαστικά τα ίδια, με μικρές διαφοροποιήσεις. Τα Pods των Διαμεσολαβητών στο εμπρόσθιο επίπεδο, τα οποία μπορεί να είναι οποιουδήποτε πλήθους, ελέγχονται από ένα Deployment, αν και, όπως έχουμε εξηγήσει νωρίτερα, θα μπορούσε να χρησιμοποιηθεί και ένα DaemonSet. Τα Pods των N_s Πρακτόρων ελέγχονται από ένα StatefulSet· αυτή τη φορά ένα εννιαίο StatefulSet για όλους τους Πράκτορες, και αυστηρά ένα Pod ανά Πράκτορα, και άρα ένα Pod ανά τεμάχιο ως προς το εύρος κλειδιών. Ο Operator, ο οποίος εκτελείται σε ένα μοναδικό Pod ανά RICAS deployment, ελέγχεται από ένα ξεχωριστό Deployment αντικείμενο.

Ένα σκαρίφημα της αρχιτεκτονικής του RICAS απεικονίζεται στο Σχήμα 27. Τα διακεκομμένα βέλη υποδεικνύουν λειτουργίες ελέγχου, ενώ τα συνεχή υποδεικνύουν λειτουργίες δεδομένων, τα οποία θα εξεταστούν παρακάτω.

Η δρομολόγηση και εκτέλεση των Pods Πρακτόρων από το Kubernetes πρέπει να λάβει χώρα με ιδιαίτερη προσοχή, κυρίως αναφορικά με τη διάταξη των στοιχείων αυτών στους φυσικούς κόμβους της υποκείμενης συστοιχίας. Για την ακρίβεια, απαιτούμε κάθε Πράκτορας να εκτελείται σε διαφορετικό φυσικό κόμβο, το οποίο επιτυγχάνεται με κατάλληλη ρύθμιση των Labels του Kubernetes [180]. Σε διαφορετική περίπτωση, δεδομένου του αλγορίθμου για την αυτοματοποιημένη δημιουργία αντιγράφων αντικειμένων που περιγράφθηκε παραπάνω, υπάρχει ο κίνδυνος τοποθέτησης δύο αντιγράφων του ίδιου αντικειμένου στον ίδιο φυσικό κόμβο. Κάτι τέτοιο θα είχε ως αποτέλεσμα το σύστημα να μην είναι ανεκτικό σε $k-1$ αποτυχίες κόμβων ως προς τη διαθεσιμότητα των αποθηκευμένων αντικειμένων, αλλά λιγότερες – το ακριβές τους πλήθος εξαρτάται από το πλήθος των αντιγράφων που έχουν τοποθετηθεί σε κοινούς κόμβους.



Σχήμα 27: *Η προτεινόμενη αρχιτεκτονική του* RICAS*, για ένα οποιοδήποτε πλήθος Διαμεσολαβητών, Πράκτορες πλήθους N_s, τον Operator, και παράγοντα αντιγραφής k.*

Οι λειτουργίες του RICAS μπορούν να ταξινομηθούν σε δύο γενικές κατηγορίες: τις λειτουργίες δεδομένων και τις λειτουργίες ελέγχου. Η πρώτη κατηγορία περιλαμβάνει τις γνωστές ενέργειες των πελατών: βασικά τη δημιουργία νέων και την ανάγνωση ήδη αποθηκευμένων μεγάλων δυαδικών αντικειμένων. Η δεύτερη κατηγορία περιλαμβάνει ενέργειες που μπορεί να προκαλέσει ο διαχειριστής του συστήματος, είτε άμεσα είτε έμμεσα. Τέτοια ενέργεια, άμεσα προκαλούμενη από τον διαχειριστή, είναι η κλιμάκωση του συστήματος, προσθέτοντας ή αφαιρώντας κόμβους, λόγου χάριν χρησιμοποιώντας το γνωστό εργαλείο `kubectl` για την τροποποίηση του ειδικού αντικειμένου του RICAS που δημιουργείται μέσω του μηχανισμού των

CustomResourceDefinition. Άλλη τέτοια ενέργεια, προκαλούμενη μόνο εμμέσως από τον διαχειριστή του RICAS, είναι η αυτόματη συλλογή σκουπιδιών που επιτελούν ταυτόχρονα όλοι οι Πράκτορες, σε συγκεκριμένες περιπτώσεις κλιμάκωσης – πρόκειται ουσιαστικά για διαγραφή αποθηκευμένων αντικειμένων για τα οποία πλέον ο εκάστοτε Πράκτορας δεν είναι υπεύθυνος, καθώς η τοπολογία του δακτυλίου συνεπούς κατακερματισμού μεταβάλλεται.

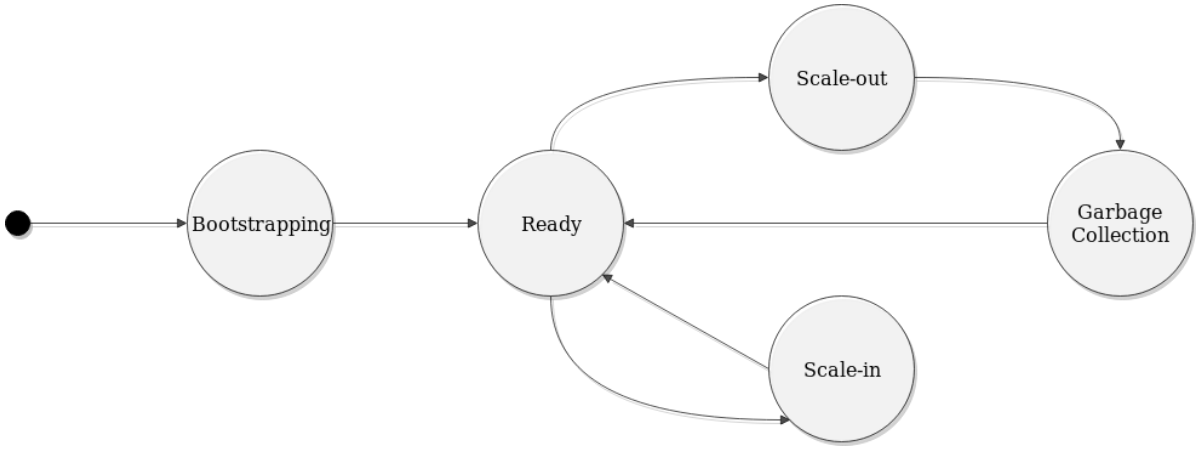
Η δεύτερη κατηγορία, οι λειτουργίες ελέγχου, έχει την απαίτηση για κατανεμημένη συμφωνία μεταξύ των Πρακτόρων, η οποία ικανοποιείται χάρη στην παροχή αισιόδοξου ελέγχου συγχρονικότητας από τη Διεπαφή Προγραμματισμού Εφαρμογών του Kubernetes, και συνιστά και τον λόγο ύπαρξης του επιπρόσθετου αρχιτεκτονικού στοιχείου: του Operator. Για να αποφευχθούν ασυνέπειες ως προς τα αποθηκευμένα δεδομένα, θα πρέπει όλοι οι Πράκτορες να έχουν κοινή αντίληψη για την ακριβή τοπολογία του δακτυλίου συνεπούς κατακερματισμού κάθε στιγμή, και να αποφασίζουν από κοινού λεπτομέρειες σχετικά με την έναρξη ή τη λήξη λειτουργιών κλιμάκωσης, και λοιπών. Ο Operator αφενός ελέγχει κατά πόσον τα αιτήματα του διαχειριστή του RICAS για λειτουργίες ελέγχου είναι έγκυρα και μπορούν πράγματι να εκτελεστούν τη στιγμή που ζητούνται, και αφετέρου συνδράμει, σε κάποιες περιπτώσεις, στην από κοινού λήψη τέτοιων αποφάσεων, με την κατά κάποιο τρόπο υψηλού επιπέδου ενορχήστρωσή τους.

Φάσεις Λειτουργίας

Οι ανωτέρω λειτουργίες ελέγχου εντάσσονται στη σχεδίαση του συστήματος με τη μορφή φάσεων λειτουργίας, όπως και στην περίπτωση του MICAS. Οι φάσεις λειτουργίας του RICAS παρουσιάζονται στο Σχήμα 28.

Οι μεταβάσεις της συστοιχίας RICAS από φάση σε φάση, πραγματώνονται βασικά με τη βοήθεια ενός ξεχωριστού συνόλου φάσεων Πράκτορα, το οποίο είναι ξεχωριστό για κάθε Πράκτορα και μοντελοποιεί το σύνολο της λειτουργίας του σε όλη τη διάρκεια της εκτέλεσής του. Το διάγραμμα φάσεων λειτουργίας κάθε Πράκτορα απεικονίζεται στο Σχήμα 29, όπου έχουν αριθμηθεί οι μεταβάσεις από φάση σε φάση, και ακολουθεί μία περίληψη της λογικής αυτών των μεταβάσεων.

1. *start* ⟶ **Ready phase**



Σχήμα 28: *Όλες οι δυνατές φάσεις λειτουργίας μίας συστοιχίας* RICAS *και οι μεταβάσεις από και προς αυτές.*

Ο Πράκτορας μεταβαίνει στη φάση Ετοιμότητας για πρώτη φορά από τη στιγμή της πρώτης εκτέλεσής του όταν η συστοιχία RICAS ολοκληρώσει τη φάση Εκκίνησής του, η οποία είναι παρεμφερής με αυτή του MICAS.

2. *start* $\longrightarrow$ **Init Sync phase**

   Ο Πράκτορας μεταβαίνει στη φάση Συγχρονισμού Εκκίνησης, κατά την οποία έλκει τα απαραίτητα δεδομένα του από τους κατάλληλους κάθε φορά Πράκτορες της συστοιχίας RICAS, στις εξής περιπτώσεις:

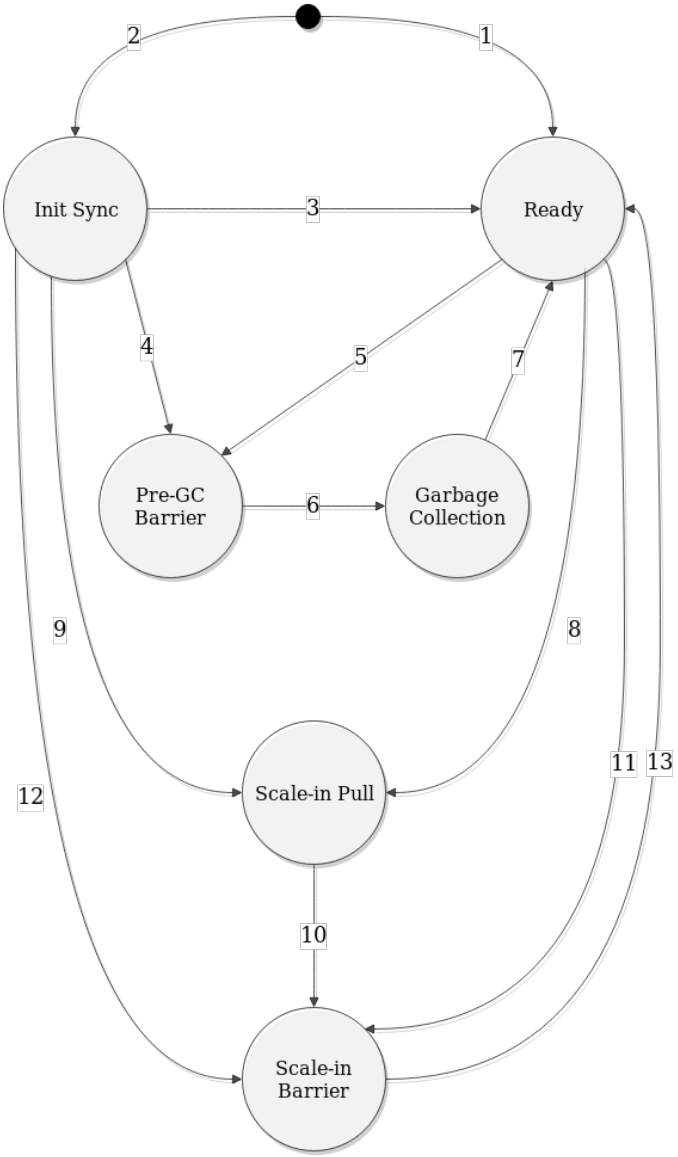
   - όταν επανεκκινεί κατά την ανάκαμψή του από σφάλμα ή αποτυχία κόμβου, ανεξαρτήτως της φάσης λειτουργίας στην οποία βρίσκεται η συστοιχία RICAS·
   - όταν εκτελείται για πρώτη φορά ως νεοεισαχθείς Πράκτορας σε προϋπάρχουσα συστοιχία RICAS ως αποτέλεσμα προηγούμενης κλιμάκωσής της για προσθήκη Πρακτόρων, και άρα σ'αυτή την περίπτωση η συστοιχία RICAS βρίσκεται στην φάση "Scale-out".

3. **Init Sync phase** $\longrightarrow$ **Ready phase**

   Ο Πράκτορας έχει μόλις ολοκληρώσει τη διαδικασία συγχρονισμού δεδομένων εκκίνησης και η συστοιχία RICAS δεν βρίσκεται εν μέσω λειτουργίας κλιμάκωσής της.

4. **Init Sync phase** $\longrightarrow$ **Pre-GC Barrier phase**

   Ο Πράκτορας έχει προσφάτως ολοκληρώσει τη διαδικασία συγχρονισμού δε-



Σχήμα 29: *Το διάγραμμα φάσεων λειτουργίας του κάθε Πράκτορα* `RICAS`*, με αριθμημένες τις μεταβάσεις από φάση σε φάση.*

δομένων εκκίνησής του, η συστοιχία `RICAS` είναι στο τελείωμα της διαδικασίας κλιμάκωσής της λόγω πρόσθεσης Πρακτόρων (φάση "Scale-out" της συστοιχίας) και πρόκειται να μεταβεί στην κατάσταση Περισυλλογής Σκουπιδιών (φάση "Garbage Collection" της συστοιχίας), και ισχύει μία από τις εξής συνθήκες:

- ο Πράκτορας έχει επανεκκινηθεί λόγω σφάλματος ή αποτυχίας κατά τη διάρκεια κλιμάκωσης της συστοιχίας με πρόσθεση Πρακτόρων, κατά την οποία ο ίδιος δεν ήταν νεοεισαχθείς·

- ο Πράκτορας είναι ένας εκ των νεοεισαχθέντων στη συστοιχία `RICAS` ως αποτέλεσμα της τρέχουσας ενέργειας κλιμάκωσής της.

5. **Ready phase ⟶ Pre-GC Barrier phase**
   Η συστοιχία `RICAS` είναι στο τελείωμα της φάσης κλιμάκωσής της μέσω προσθήκης νέων Πρακτόρων (φάση "Scale-out" της συστοιχίας), και πρόκειται να μεταβεί στη φάση Περισυλλογής Σκουπιδιών (φάση "Garbage Collection" της συστοιχίας), κατά την οποία κλιμάκωση ο εν λόγω Πράκτορας δεν ήταν νεοεισαχθείς στη συστοιχία `RICAS`, και δεν είτε δεν είχε κανένα κώλυμα λόγω σφάλματος ή αποτυχίας, είτε κάποιο ενδεχόμενο τέτοιο κώλυμα αντιμετωπίστηκε πλήρως νωρίτερα.

6. **Pre-GC Barrier phase ⟶ Garbage Collection phase**
   Ο Πράκτορας είναι ενήμερος ότι όλοι οι υπόλοιποι Πράκτορες της συστοιχίας `RICAS` έχουν φτάσει σε ένα "κατανεμημένο φράγμα" πριν τη φάση Περισυλλογής Σκουπιδιών, και συνεπώς μπορεί να προχωρήσει στην περισυλλογή των τοπικών του σκουπιδιών, δηλαδή των αντικειμένων που δεν χρειάζεται πλέον να αποθηκεύει ο ίδιος.

7. **Garbage Collection phase ⟶ Ready phase**
   Ο Πράκτορας έχει μόλις ολοκληρώσει την τοπική του περισυλλογή σκουπιδιών, και έχει ενημερώσει τους υπόλοιπους Πράκτορες μέσω του προσαρμοσμένου αντικειμένου του `RICAS` της Διεπαφής Προγραμματισμού Εφαρμογών του Kubernetes που δημιουργήθηκε μέσω `CustomResourceDefinition`. Η συστοιχία `RICAS` ήταν στη φάση Περισυλλογής Σκουπιδιών, και μπορεί να μεταβεί στη φάση Ετοιμότητας αν ο εν λόγω Πράκτορας είναι ο τελευταίος που πραγματοποιεί αυτή του τη μετάβαση.

8. **Ready phase ⟶ Scale-in Pull phase**
   Ο Πράκτορας – ο οποίος, εφόσον βρίσκεται στη φάση Ετοιμότητας, δεν παρουσιάζει τη στιγμή αυτή κάποιο κώλυμα στη φυσιολογική του λειτουργία – έχει μόλις παρατηρήσει ένα εκκρεμές αίτημα για κλιμάκωση της συστοιχίας `RICAS` μέσω αφαίρεσης Πρακτόρων, κατά την οποία ο ίδιος θα παραμείνει εντός της. Συνεπώς, μεταβαίνει από τη φάση Ετοιμότητας στη φάση "Scale-in Pull" για να ξεκινήσει τη διαδικασία έλξης των νέων δεδομένων που πιθανώς να του αναλογούν βάσει της νέας τοπολογίας του δακτυλίου συνεπούς κατακερματισμού.

9. **Init Sync phase ⟶ Scale-in Pull phase**

   Ο Πράκτορας έχει μόλις ολοκληρώσει τη διαδικασία συγχρονισμού δεδομένων εκκίνησης και παρατηρεί ότι υπάρχει ένα εκκρεμές αίτημα κλιμάκωσης της συστοιχίας `RICAS` μέσω αφαίρεσης Πρακτόρων, κατά τη διάρκεια της οποίας ο ίδιος θα παραμείνει εντός της. Η αποτυχία, λόγω της οποίας ο εν λόγω Πράκτορας είχε επανεκκινηθεί, μπορεί να έχει συμβεί είτε πριν είτε αφού έλαβε χώρα το αίτημα κλιμάκωσης. Ο Πράκτορας αυτός, μεταβαίνει από τη φάση Συγχρονισμού Εκκίνησης στη φάση "Scale-in Pull" ώστε να ξεκινήσει την έλξη των νέων δεδομένων που ενδεχομένως να του αναλογούν βάσει της νέας τοπολογίας του δακτυλίου συνεπούς κατακερματισμού, αμέσως αφού έχει έλξει όλα τα δεδομένα που του αναλογούσαν βάσει της παλαιάς τοπολογίας του δακτυλίου συνεπούς κατακερματισμού (προτού δηλαδή ξεκινήσει η διαδικασία κλιμάκωσης του συστήματος). Η συστοιχία `RICAS` βρίσκεται στη φάση "Scale-in", δηλαδή εν μέσω κλιμάκωσής της λόγω αφαίρεσης Πρακτόρων.

10. **Scale-in Pull phase ⟶ Scale-in Barrier phase**

    Ο Πράκτορας έχει μόλις ολοκληρώσει τη διαδικασία συγχρονισμού των δεδομένων του λόγω κλιμάκωσης της συστοιχίας `RICAS` μέσω αφαίρεσης Πρακτόρων – ενώ η συστοιχία βρίσκεται ακόμα στη φάση "Scale-in" – και μεταβαίνει στη φάση "Φράγματος Scale-in", όπου αναμένει τους υπόλοιπους Πράκτορες να ολοκληρώσουν και τις δικές τους διαδικασίες συγχρονισμού.

11. **Ready phase ⟶ Scale-in Barrier phase**

    Ο Πράκτορας – ο οποίος δεν βρίσκεται εν μέσω κάποιου κωλύματος ή συγχρονισμού δεδομένων λόγω πρότερης αποτυχίας ή σφάλματος – έχει μόλις παρατηρήσει ότι η συστοιχία `RICAS` μεταβαίνει στη φάση κλιμάκωσής της μέσω αφαίρεσης Πρακτόρων (φάση "Scale-in" της συστοιχίας), κατά την οποία ο ίδιος αποχωρεί από τη συστοιχία. Σε αντίθεση με την περίπτωση των Πρακτόρων που παραμένουν εντός της συστοιχίας κατά την κλιμάκωση, ο εν λόγω Πράκτορας δεν χρειάζεται να έλξει νέα δεδομένα, αφού ο ίδιος δεν συμπεριλαμβάνεται στη νέα κατάσταση του δακτυλίου συνεπούς κατακερματισμού. Παρ'ολ'αυτά, πρέπει να περιμένει όλους του παραμένοντες Πράκτορες να ολοκληρώσουν τη διαδικασία συγχρονισμού των δεδομένων τους βάσει της νέας αυτής κατάστασης του δακτυλίου, ώστε να διασφαλιστεί η υψηλή διαθεσιμότητά τους σε πε-

ριπτώσεις αποτυχίας των Πρακτόρων. Έτσι, ο εν λόγω Πράκτορας μεταβαίνει απευθείας από τη φάση Ετοιμότητας στη φάση "φράγματος Scale-in", όπου και παραμένει εξυπηρετώντας εισερχόμενα αιτήματα συγχρονισμού δεδομένων τα οποία καταφθάνουν από τους παραμένοντας μετά την κλιμάκωση Πράκτορες, αλλά και πιθανά αιτήματα ανάγνωσης ή εγγραφής δεδομένων που ενδεχομένως καταφθάσουν από τους Διαμεσολαβητές.

12. **Init Sync phase $\longrightarrow$ Scale-in Barrier phase**
    Ο Πράκτορας είναι ένας από τους αποχωρήσαντες μιας τρέχουσας διαδικασίας κλιμάκωσης της συστοιχίας `RICAS` μέσω αφαίρεσης Πρακτόρων, και ως τέτοιος έχει προσφάτως επανεκκινηθεί ως αποτέλεσμα κάποιου σφάλματος ή αποτυχίας, η οποία μπορεί να συνέβη είτε πριν είτε αφού κατατέθηκε το αίτημα για κλιμάκωση της συστοιχίας. Αν καθένας από τους παραμένοντες Πράκτορες έχει ολοκληρώσει τη διαδικασία συγχρονισμού των δεδομένων του λόγω της κλιμάκωσης, ο εν λόγω Πράκτορας δεν έχει λόγω να ξεκινήσει καν τη δικιά του, και απλά την προσπερνά, μεταβαίνοντας στη φάση "φράγματος Scale-in", ώστε να λάβει μέρος στην επιδίωξη των υπολοίπων να ενημερώσουν το `StatefulSet` για τη νέα κατάσταση της συστοιχίας. Διαφορετικά, αν δηλαδή υπάρχει οποιοδήποτε πλήθος παραμένοντων μετά την κλιμάκωση Πρακτόρων οι οποίοι δεν έχουν ολοκληρώσει τη διαδικασία συγχρονισμού των νέων δεδομένων που πιθανώς να τους αναλογούν, τότε ο εν λόγω αποχωρήσας Πράκτορας ξεκινά τη δική του διαδικασία συγχρονισμού δεδομένων εκκίνησης ώστε να ανακτήσει όλα τα δεδομένα που του αναλογούν βάσει της κατάστασης του δακτυλίου συνεπούς κατακερματισμού πριν την κλιμάκωσή του, προκειμένου να είναι σε θέση στο μέλλον να εξυπηρετήσει και ο ίδιος εισερχόμενα αιτήματα συγχρονισμού δεδομένων που ενδεχομένως καταφθάσουν από αυτούς τους παραμένοντες μετά την κλιμάκωση Πράκτορες.

13. **Scale-in Barrier phase $\longrightarrow$ Ready phase**
    Ο Πράκτορας έχει επιβεβαιώσει ότι το `StatefulSet` των Πρακτόρων `RICAS` έχει ενημερωθεί για την πρόσφατη κλιμάκωση μέσω αφαίρεσης Πρακτόρων, και ότι η λειτουργία των αποχωρήσαντων Πρακτόρων πρόκειται να τερματιστεί πολύ σύντομα. Η συστοιχία `RICAS` είτε έχει ήδη μεταβεί στη φάση Ετοιμότητας, είτε πρόκειται να μεταβεί πολύ άμεσα. Αν ο εν λόγω Πράκτορας είναι ένας εξ

αυτών που παραμένουν στη συστοιχία μετά το πέρας της διαδικασίας κλιμάκωσης, συνεχίζει τη φυσιολογική λειτουργία του. Διαφορετικά, αν δηλαδή είναι ένας εξ αυτών που αποχωρούν από τη συστοιχία, μεταβαίνει και πάλι στη φάση Ετοιμότητας, περιμένοντας το τοπικό του kubelet να τον ενημερώσει για την έναρξη της διαδικασίας ομαλού τερματισμού λειτουργίας του.

1 Introduction

To begin with, we outline the scope of our work. First, we provide a quick overview of the situation at hand, highlighting the factors that render it particularly interesting nowadays. Next, we adumbrate the flow of our reasoning on confronting the problem, along with our respective design attempts. After this, we move on to describe some of the existing solutions in brief, examining parameters that might be making some of them rather cumbersome to work with, and some others to potentially thrive. Finally, we present how various concepts, as well as design and implementation issues are structured within this thesis.

1.1 Motivation

1.1.1 The State of the Datacenter

We live in the, so called, Information Age; multiple facets of economic and social growth are largely driven by the Information industry. Individuals, as much as corporations, produce and hand in data to the latter, and rely on it to deliver solutions for their personalized needs. In the most recent years, we are witnessing a historic peak of the phenomenon, which, from a technological point of view, raises new hurdles that revolve around the vast amount of data and the necessity of their efficacious handling.

This sort of impediments spans over most of the classic constituents of computing systems: processing, networking and storage, and have brought about sets of solutions that leverage the theory of distributed computing since only recently, albeit the latter

being present and evolving for several decades in the literature. Together they shift the paradigm in data processing and management to applications based on distributed systems, shaping a whole new field of exploration and optimizations, often associated with the "Big Data".

The infrastucture required to accommodate these distributed systems and the volume of information that is either stored or flows in and out of them, usually consists of datacenters, in some cases large, and maybe even multiple interconnected ones, dispersed around the globe. In addition, to both lower the development cost and avoid the danger of vendor lock-ins, the ongoing trend of commoditization of the hardware infrastructure increasingly gains ground.

In the meantime, the fast-paced emergence of Microservices patterns in software architecture has inadvertently led to a perpetual re-evaluation of the application deployment model. The use of virtual machines, erstwhile the default choice as deployment units in datacenters, is gradually superseded by, or intermixed with, containers. This new piece of technology, which is based on concepts and constructs that have been present for many years but popularized by Docker, Inc. (then dotCloud, Inc. [1]) and the Docker Project[1] circa 2013, rises today as the de facto standard for this purpose across the industry. As one of the latest trends in the field, innovative systems are actively being developed to underpin it, forming a new ecosystem with promising features around it.

While more organizations take on the journey to containerization, day-by-day, the mutual need for efficient tooling has become prevalent, hence occupies a big portion of the arising ecosystem. Open-source systems like fleet[2] and Container Linux[3] (formerly CoreOS Linux [2]) by CoreOS, Inc. (now acquired by Red Hat [3]), and Kubernetes[4], born at Google, were among the first container-specific open-source deployment and orchestration tools incepted. Since their release and the recognition of their value in production, the variety of available options has been greatly improved, and many more systems, both open-source and commercial, have been either built, anew or atop exist-

[1] Ben Golub, October 2013, *dotCloud, Inc. is Becoming Docker, Inc.*, `https://blog.docker.com/2013/10/dotcloud-is-becoming-docker-inc/`, accessed on April 24th, 2018.

[2] Alex Polvi, December 2016, *Self-Driving Kubernetes, Container Linux by CoreOS and Kubernetes 1.5*, `https://coreos.com/blog/tectonic-self-driving.html`, accessed on April 24th, 2018.

[3] Alex Polvi, January 2018, *CoreOS to join Red Hat to deliver automated operations to all*, `https://coreos.com/blog/coreos-agrees-to-join-red-hat/`, accessed on April 24th, 2018.

ing ones, or retrofitted to support containers, such as Tectonic[5] by CoreOS, Docker Swarm[6] by Docker, OpenShift[7] by Red Hat, Apache Mesos[8], DC/OS[9] by Mesosphere, and many others, as this list is not intended to be comprehensive.

In this thesis, we focus entirely on Kubernetes[4] over Docker[1]. A huge amount of effort is being put into developing the former as the standard platform for deploying and managing containerized applications on cloud environments, either on-premises or third party hosted. Its community is constantly growing, with more and more developers joining it as the time goes by, either independently or through companies, including some industry giants like Google, Red Hat, Microsoft, IBM, Intel, VMWare, Alibaba, Huawei, HP, Cisco Systems, Samsung SDS, Oracle, AT&T, and others. It is our belief that this is not just an evanescent fuss, and that Kubernetes is soon to be the de facto established tool for this goal. We postpone a quick overview of its functionality till chapter 2.

1.1.2 The State of State

Service statelessness originates from the design principles of Service Oriented Architecture. According to [10], *"a service can be active but may not be engaged in the processing of state data. In this idle condition, the service is considered to be stateless. As you may have guessed, a service that is actively processing or retaining state data is classified as being stateful"*. The same concept is, more or less, pertained in Microservices Architecture design. In other words, stateless applications are microservices that do not process or store persistent data, despite possibly storing temporary data; typical examples being the many kinds of web applications, like frontend UIs[4], that often use the HTTP[5] protocol, which itself is stateless by design. On the other hand, stateful applications are such that keeping state is critical to running the service, and therefore some sort of persistent storage is required; databases and any transaction-based microservices are indicative of this category. Another accurate distinction of the two kinds of services is presented in [11], according to which, stateless services may be *"services which aggregate responses from other services"*, whereas a stateful service is *"a service that generates its response by executing business logic on its state stored in persis-*

[4]User Interfaces.
[5]Hypertext Transfer Protocol.

tent store".

One of the handiest qualities of stateless applications is how effortlessly scalable they tend to be. Typically, they can be scaled horizontally merely by spinning up more instances behind a simple round-robin load balancer, while ephemeral low-capacity storage suffices for their entire functionality. This is aligned with container runtime engines' common underlying architecture (e.g. [12]). A container is comprised of multiple immutable read-only layers, which form the "image", and any number of them may be shared by multiple containers. A single, thin, writable layer on top of them is where any temporary data modifications are stored; each container has its own writable layer, which is deleted alongside the deletion of the container. It is obvious that stateless applications turn out to be the perfect candidates for containerized deployments; indeed, in many cases they have been the entry point for organizations to experiment with this new deployment model.

A major problem, though, is that, as StorageOS[6] poses it through the words of Alex Chircop in [13], *"there's no such thing as a stateless architecture"*. An overwhelming majority of services need to store state information *somewhere*, if to do anything meaningful at all, so there has to be at least one architectural component that deals with the need for dedicated persistent storage. But how is that supposed to be dealt in the modern containerized world of microservices, so far? As Portworx Annual Container Adoption Survey 2017[14] has revealed, among various other interesting facts, persistent storage for containers remains one of the hardest problems yet to be truly solved. As a matter of a fact, even The Twelve-Factor App[15], a widely popular methodology for developing modern cloud-native applications, conveniently groups all stateful services, including databases and messaging, queuing and caching systems, under the term "Backing Services", and suggests treating them as attached resources, effectively separating them from a pure cloud-native design.

It is also worth noting that stateful applications are inherently harder to scale horizontally in comparison with stateless ones, especially those originally designed for a non-containerized world, such as traditional database management systems, and require special treatment, often individualized to each case. Each instance usually has a unique identity and a specified role in the system, in the sense that multiple instances

[6]Storageos Ltd, `https://storageos.com/company/`, accessed on April 24th, 2018.

are not interchangeable. This identity can be tied, for example, to a specific storage entity (e.g. a volume) over which the application may need to perform some kind of recovery upon restarting from failures, or it can be linked to a specific role in a distributed protocol (e.g. to provide distributed concensus).

1.1.3 Distributed Storage Systems

Prior to the explosion of containerization, driven by the commoditization of the hardware infrastructure but also enhancing its consolidation, another movement, that of software-defined storage, was already prevailing.

Traditional storage solutions in the datacenter were delivered via SANs[7]. The use of proprietary software stacks running on top of proprietary hardware appliances, consisted of multiple disk arrays, controllers and disk trays has been the norm, and perhaps still is, in the enterprise market. However, even though these solutions have also been continuously evolving and striving to adapt to the contemporary needs of Big Data architectures, it is too expensive and difficult for organizations to adjust to the exponential data growth using them.

As an alternative, a plethora of novel distributed storage systems have been developed during the last decade, and some of them have actually proven themselves to be remarkably robust and reliable for use in production. Rather than relying on faster drives or higher bandwidth, they employ API[8]-centric administration techniques for programmatically performing traditional storage-related tasks, like provisioning, management and monitoring, over commodity disks. With the latter being cheap, easily replaceable, increasingly performant (e.g. NVM[9] technology, like SSDs[10], and since recently, in combination with NVMe[11]), and most importantly promoting the independence from proprietary platforms, these systems achieve data consistency and redundancy, as well as high availability even in cases of network partitions, using sophisticated algorithms and distributed computing concepts.

Instead of attempting to list the most notable among them, which would anyway raise

[7] Storage Area Networks.
[8] Application Programming Interface.
[9] Non-volatile memory.
[10] Solid-State Drives.
[11] NVM Express.

a controversial point since quite many of them introduced features that influenced the rest, we choose to only refer to two of them that also served as a source of inspiration for our own design: Ceph[16, 17, 18] and Cassandra[19, 20].

1.2 Problem Statement

In this thesis, we focus on the interoperability of containers and local peristent storage in the modern world of cloud-native microservices.

We argue that distributed storage systems should be taking full advantage of the multitude of benefits yielded by the use of containers for developing, packaging and deploying applications on cloud infrastructure, and the convenience from the features provided by the orchestration tools of their thriving ecosystem.

Our primary objective is to examine how persistent storage can be accomplished by the means of local disks and natively using Kubernetes and Docker as the platform. We explore how the exposed API of the former may be of service to a set of "intelligent" containerized agents and proxy servers, allowing them to undergird some of the quintessential characteristics of modern distributed storage systems, namely **incremental scalability** and **elasticity**, **load-balancing**, **self-healingness**, **high availability** and **fault tolerance** via **data replication**.

To this end, we design and implement a simple, yet sound, distributed object store, adhering to the principles of Microservices architecture. To mitigate the massive complexity introduced by update operations regarding the consistency of the data stored in distributed storage systems, without deviating from the original purpose of the thesis, we leverage the concepts of **data immutability** in conjunction with **content-addressability**, **hashing** and **sharding**, which are all discussed in Chapter 3.

The state is stored as binary objects (i.e. blobs[12]) identified by unique keys. The object's key is the sole access method available; sticking by the simplified design promoted by the NoSQL[13] model, in contrast to traditional RDBMSs[14], complex querying, indexing and management functionality are deemed excess for our purpose, and thus are not

[12] Binary large objects.
[13] Not-only SQL.
[14] Relational Database Management Systems.

supported.

On the first design iteration, our attempt is based on a **static sharding** scheme. Inspired by Ceph's RADOS[15][17], the keyspace is partitioned into a static number of shards, which are mapped onto sets of containers. The corresponding shard of each read or write request for an object, incoming to a reverse-proxy server on the frontend, is calculated by applying a very simple hash function on the object's key. The request is then routed to the appropriate set of agents on the backend. This is akin to the distribution of objects among a set of Placement Groups conducted by RADOS using the CRUSH[16][18] algorithm.

Following some scalability issues of the design above, mostly related to Kubernetes, on the second design iteration our attempt employs a **consistent hashing**[21] algorithm among the agents on the backend, while preserving the rest of the system's architecture as intact as possible. Consistent hashing has been put into practice before by a number of distributed storage systems, including, but not limited to, Amazon's Dynamo[11], Cassandra[19, 20], Openstack's Object Storage service Swift[22], Basho's Riak KV[23], Voldemort[24], GlusterFS[25], hence is regarded as a reliable, battle-tested technique for this kind of difficulties.

The final result is a highly available, linearly scalable, self-healing, cloud-native architecture for a replicated distributed object store. Currently being implemented as a Kubernetes API extension, simply attaching a proper local storage engine would allow it to operate as a decent standalone distributed object store for simple use cases. Otherwise, it can be seen as an intermediate layer of a complete distributed storage system that would also involve a higher layer on top of it, to manage object versioning, enabling update and other complex operations.

1.3 Existing Solutions

The problem of using local commodity hardware to provide persistent storage to stateful applications deployed on Kubernetes is not introduced for the first time by this thesis. Multiple attempts have taken place and many projects and products, actively and

[15]Reliable, Autonomic Distributed Object Store.
[16]Controlled Replication Under Scalable Hashing.

rapidly evolving, are confronting it. Nevertheless, none of them has managed to hit the nail on the head to the point of standardizing a solution, so far.

Most of the existing solutions are attempts to adjust the deployment procedure of existing distributed storage systems to the terms imposed by Kubernetes. Indeed, the robustness and the reliability provided by systems like Ceph and Cassandra, which have been present and tested in production use cases for more than a decade, is an alluring trait for a solution to a problem of such importance, as storage is. However, there has also been at least one serious effort to build a "Kubernetes-native" distributed storage system from scratch, whose vision governs the attempts of this thesis as well.

In this section, we briefly describe some of the most notable existing solutions for our problem.

1.3.1 Torus

Torus[26] is an open source distributed storage system designed by CoreOS to provide reliable, scalable storage to container clusters orchestrated by Kubernetes. As one of its authors describe it, at its core, it is a library with an interface that appears as a traditional file, allowing for storage manipulation through well-understood basic file operations [27]. It also supports exposing this file as block-oriented storage via NBD[17].

Torus employs consistent hashing to support replication, garbage collection, and pool rebalancing through an internal peer-to-peer API. The design also enables support for encryption and Reed-Solomon error correction[31], to provide greater assurance of data validity and confidentiality throughout the system.

Any data concerning the "state of the world", which usually require consensus among the peers are being run through etcd[28]. These include discovery of peers, coordination, checkpointing, metadata for keeping track of volumes, as well as any other metadata, but do not include the bulk of stored users' data itself [29]. In Torus' architecture, etcd is the "Metadata Store".

According to [29], other than the Metadata Store, Torus is structured roughly in three layers. The *Server* is the upper layer that users of the system interact with, to store and

[17]Network Block Device.

retrieve data. It is also responsible for heartbeats and monitoring, thus it communicates with etcd. Below that, the *Distributor* layer handles the peer-to-peer communication via gRPC[30], which is essential to both rebalancing as well as garbage collection. It also keeps track of the kind of storage that lies underneath it, in the *Storage* layer, which is more or less a key-value store where the actual data blocks live.

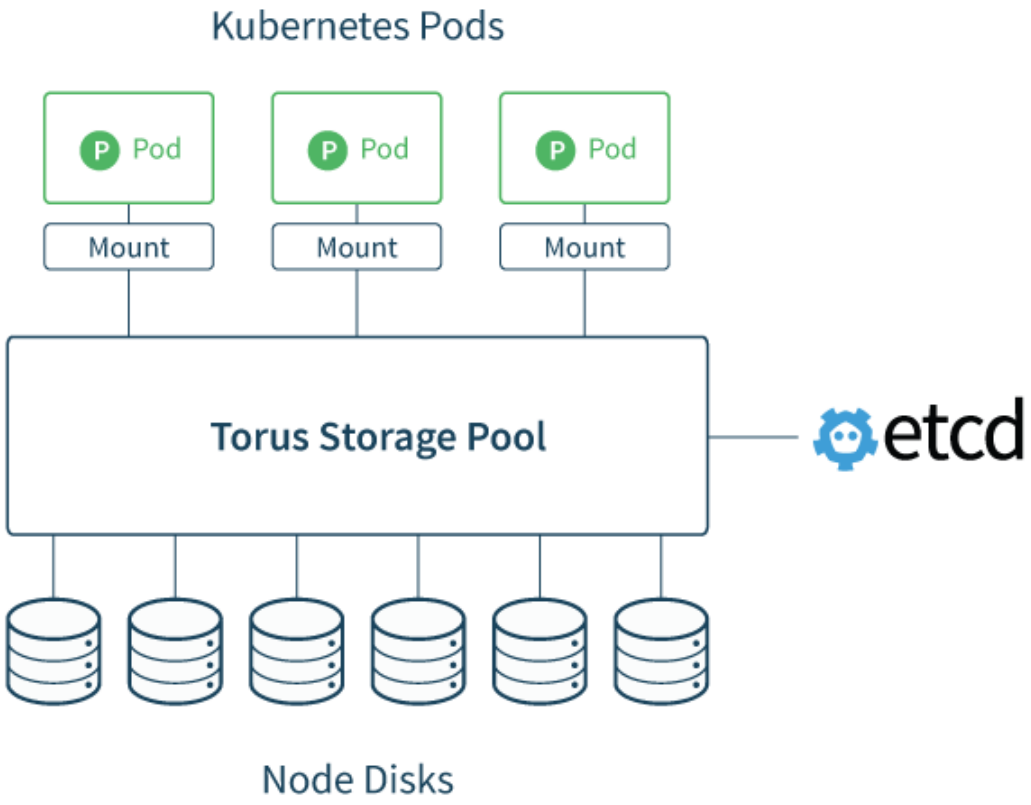


Figure 1.1: *Torus' model for providing local persistent storage to Kubernetes Pods. [26]*

According to [26], Torus started as a prototype in June 2016 to build a storage system that could be easily operated on top of Kubernetes, and that model has been proven out, but neither the development velocity nor the depth of community engagement were satisfactory. Therefore, the development of Torus by CoreOS has been discontinued since of Feb 2017. Although the community is encouraged to fork and continue the project, no such significant endeavor has come to our attention.

1.3.2 Rook: Ceph on Kubernetes

Rook[32] is a CNCF[18] sandbox project that aims to automate the deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management of existing distributed storage systems using the facilities provided by an underlying cloud-native container management, scheduling and orchestration platform. It is currently in alpha state and has focused initially on orchestrating Ceph on top of Kubernetes, using its primitives.

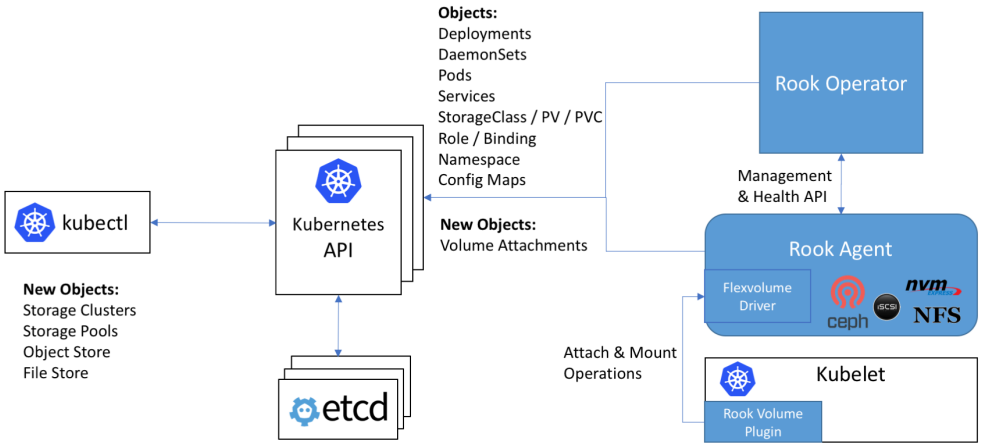


Figure 1.2: *Rook's integration with Kubernetes. [32]*

With Rook running in the Kubernetes cluster, Kubernetes applications can mount block devices and filesystems managed by Rook, or can use the S3[19]/Swift API for object storage. The Rook operator is a simple container that starts and monitors Kubernetes Pods running Ceph Monitors and a Kubernetes DaemonSet running the Ceph OSDs[21] that provide basic RADOS storage. It also manages Kubernetes CRDs[22] for pools, object stores (S3/Swift), and file systems by initializing any Kubernetes arti-

[18]Cloud Native Computing Foundation, `https://www.cncf.io/`, accessed on April 24th, 2018.

[19]Amazon Web Services' Simple Storage Service, or AWS S3, is a public cloud platform for storing binary or unstructured data [33]. Its REST[20]API, usually over HTTP, is widely popular across the industry when it comes to object storage.

[20]Representational State Transfer.

[21]Object Storage Daemons.

[22]Custom Resource Definitions.

facts necessary to run the services, and monitors the storage daemons to ensure the cluster is healthy. The Ceph Monitors are started or failed over when necessary, and other adjustments are made as the cluster grows or shrinks. The Rook operator also deploys an agent on every Kubernetes node, which configures a Kubernetes Flexvolume plugin that integrates with Kubernetes' volume controller framework, and allows the required storage operations to be handled, for example attaching network storage devices, mounting volumes and formating the filesystem.

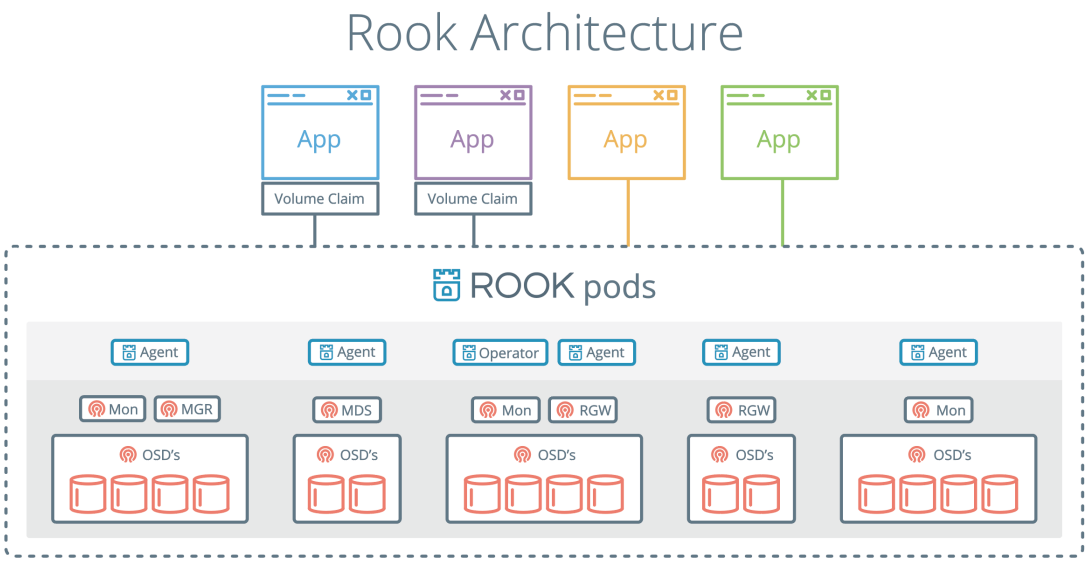


Figure 1.3: *Rook's architecture for Ceph on top of Kubernetes. [32]*

Rook does not attempt to maintain full fidelity with Ceph. It includes all necessary Ceph daemons and tools to manage and store all data, but many of the Ceph concepts, like placement groups and crush maps, are hidden. Instead, it creates a much simplified user experience for administrators, which is in terms of physical resources, pools, volumes, filesystems, and buckets, while advanced configuration can still be applied, if needed, using the Ceph tools.

Rook is way more complete of a project compared to the resulting system of this thesis, as serious effort has been put into its design and development to meet the standards for use in production environments. This is not the case for our system; as mentioned previously, we have only focused on a fraction of the necessary features for a distributed storage system to be production ready, and we have not tried to attach an appropriate storage engine underneath, yet.

1.3.3 Cassandra on Kubernetes

Since the beginnings of its design and implementation, Cassandra has been supporting replication of the data it stores, taking into consideration the actual topology of the nodes that it is deployed on, at both rack and datacenter level. In order for this to be feasible, Cassandra has been designed based on the presumption of bare-metal deployments, with every Cassandra instance being tied to its underlying physical server. This is obviously contradictory to the use of cluster scheduling and orchestrating systems, like Kubernetes, which treat the physical servers as "cattle" instead of "pets"[34] and containers as lightweight and utterly portable entities across hosts, thus rendering the containerized Cassandra deployment a non-trivial issue in cases that its full functionality is required.

However, there is a number of solutions that perform quite well in some cases. Not all of them are equivalent, as they use different techniques for handling the persistence layer, and some of them may still fail to leverage Cassandra's rich set of features to the fullest. Covering them extensively or comparing them is out of the scope of this work, yet we ought to mention some of them for the sake of completeness. We emphasize that this is not even a summary of the cited products' functionality; it is merely a non-exhaustive list of options about our specific situation, and the agog reader is urged to dive into the respective sources for more information.

1.3.3.1 Native Kubernetes deployment

Kubernetes' official documentation pages include an example of a containerized Cassandra deployment[35], showing how the associated requirements and concepts are mapped onto the Kubernetes primitives. However, the provided example only makes use of Minikube's[36] default storage provisioner, which is not suitable for use in production. Based on the above, IBM has published a more recent guide[37] that also includes provisioning of local storage volumes via Kubernetes (support for which, by the way, was added only in recent Kubernetes releases).

1.3.3.2 Deployment using PX by Portworx, Inc.

According to [38], PX, Portworx's proprietary product, is a shared-nothing, loosely distributed block storage layer, deployed as a containerized service to a cluster's nodes. It detects the available hardware (e.g. type of drives, capacity, or any other capabilities), and thus makes the whole PX cluster aware of the overall topology of the participating nodes in the datacenter (or across many datacenters), and so remains, using a gossip protocol[39]. It creates a global storage pool available across all nodes that provides the actual physical capacity for volumes, thinly provisioned via orchestrators like Kubernetes, with which PX is integrated. PX is then in the data path, strategically placing the blocks written on the various servers, pursuing high availability of the data.
The company provides a guide[40] for deploying Cassandra with Portworx.

1.3.3.3 Deployment using StorageOS by StorageOS, Inc.

StorageOS's operation looks very similar to the above, according to [41]. It is deployed to a cluster's nodes to figure out the overall topology and capabilities of the cluster, and the StorageOS nodes communicate with each other using a gossip protocol to maintain a consistent view of the state of the cluster. A highly available storage pool is also created, that backs the volumes that are thinly provisioned via orchestrating systems like Kubernetes, with which StorageOS is integrated. StorageOS is also in the data path, transparently redirecting reads and writes from the containers to the appropriate volumes.
The company provides a guide[42] for deploying Cassandra with StorageOS.

1.3.3.4 Deployment using Rok by Arrikto, Inc.

Arrikto presents an alternative approach through Rok[43, 44]. Cassandra can be deployed on persistent storage volumes provisioned via Kubernetes, with which Rok is fully integrated. Instead of being constantly in the data path, and thus unlike Portworx's PX and StorageOS, Rok periodically produces snapshots of the data (efficiency is achieved via incremental snapshotting), and only intervenes during node recovery, producing a thinly-cloned volume from the snapshot data, transparently to the applications consuming them. Rok significantly accelerates the recovery of the data by

performing it on the block level for as many data as possible (i.e. for the data of the most recent snapshot), before Cassandra's repairing mechanism kicks in to handle any changes in the data that occured since the most recent snapshot. In coordination with Kubernetes, other than the automated and efficient backup, recovery and migration of Cassandra nodes, it enables data mobility across multiple administrative domains, thus fully restoring the otherwise impaired Cassandra's functionality due to containerization, while preserving a purely cloud native experience.
The company provides a comparison of Cassandra deployment solutions including local NVMe storage using Rok and other commercial external storage services in [45].

1.4 Thesis Structure

The content of the thesis is organized in multiple chapters and structured as follows:

- **Chapter 2**: a brief overview of some of the core concepts and systems that our work is founded upon.
- **Chapter 3**: a thorough analysis of critical decisions regarding the design and architecture of our systems, along with concepts, algorithms and techniques that have been either selected or developed to support or enhance them.
- **Chapter 4**: a brief demonstration of some of the focal points of our development process, including mainly details about the implementation of certain design choices that could have been implemented in multiple different ways.
- **Chapter 5**: concluding remarks and future improvements and extensions to render the proposed systems robust in demanding environments.

2 Background

In this chapter we provide some elementary information regarding various aspects of the background knowledge required to understand the design and the implementation of the distributed storage systems proposed in this work. We begin by deconstructing containers and their implementation in the Linux kernel into their principal constituents: Linux namespaces and control groups. After briefly describing the functionality of these components, we discuss about union filesystems. Next, we demonstrate how all these concepts and mechanisms can be used to cooperatively formulate the foundation for a container runtime engine: we discuss about some aspects of Docker and its take on the design and implementation of such a system. We move on to concisely analyze some facets of Kubernetes and its rationale: its perspective on containerization, some software architecture and engineering choices, several abstractions that it creates, and more. This chapter concludes with a quick presentation of rsync, a file transfer utility that employs a unique algorithm to achieve efficiency, mostly in terms of network bandwidth utilization.

At this point, we have to remark that this chapter's content is by no means a complete analysis or explanation of all concepts and systems involved in the thesis. It rather is a high-level overview of the basic functionality and architecture of some of them. Therefore, this chapter is safe to skip for the readers who are already familiar with the topics mentioned in the above paragraph, whereas it only provides elementary, high-level information about them to the readers who are not already familiar with them, since the details are deliberately missing.

Additional background concepts and knowledge that may be required for fully understanding the reasoning of the arguments in the thesis are sometimes presented in the following chapters as well. Such concepts may constitute design or implementation decisions, thus fitting better in chapter 3 or chapter 4.

2.1 Containers in Linux

Fundamentally, containers are isolated user-space instances that can be created thanks to *operating-system-level virtualization*. Ordinary operating system processes are normally aware of all the resources of the host machine they are running on, such as files and directories, connected devices, CPU and memory resources, etc. Operating system processes running inside a container are aware only of what the container engine permits them to see, and they have the illusion that they are the only processes running on the system.

In Unix-like operating systems, such as Linux, the isolation provided by containers resembles an "advanced implementation" of the standard `chroot`[47] mechanism. However, the actual isolation mechanism behind containers in the Linux kernel is known as **Linux namespaces**. In addition to the isolation mechanism, the Linux kernel also provides a resource-management mechanism that allows limiting the impact of one container's activities on other containers running on the same host, which is known as **control groups**.

2.1.1 Linux Namespaces

In Linux, there are originally six different types of namespaces that are currently implemented, plus a seventh one implemented more recently. The purpose of each namespace is to wrap a global system resource in an abstraction, which makes it appear to the processes within the namespace as if they had their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes [48]. One of the overall goals of namespaces is to support the implementation of operating-system-level virtualization.

2.1.1.1 Namespaces API

The Linux kernel provides an API for manipulating namespaces, which consists of a system call dedicated to them, a set of flags that can be passed as arguments to other system calls with a broader purpose, as well as various `/proc` files[49].

- `clone(2)` [48, 50]
  The `clone(2)` system call originally forks off new processes, and also implements a number of features unrelated to namespaces. If the `flags` argument of the call specifies one or more of the `CLONE_NEW*` flags, which will be mentioned later, then new namespaces are created for each flag, and the child process is made a member of those namespaces.

- `unshare(2)` [48, 51]
  The `unshare(2)` system call, among other things (not necessarily related to namespaces), moves the calling process to a new namespace. If the `flags` argument of the call specifies one or more of the `CLONE_NEW*` flags, then new namespaces are created for each flag, and the calling process is made a member of those namespaces.

- `setns(2)` [48, 52]
  The `setns(2)` system call allows the calling process to join an existing namespace. The namespace to join is specified via a file descriptor that refers to one of the `/proc/[PID]/ns` files, which are discussed below.

- `/proc/[PID]/ns/` [48, 49]
  Each process has a `/proc/[PID]/ns/` subdirectory that contains one entry for each namespace that supports being manipulated by `setns(2)`. *Bind mounting*[53, 54] one of the files in this directory to somewhere else in the filesystem keeps the corresponding namespace of the process specified by `PID` alive, even if all processes that are currently in the namespace terminate. Opening one of the files in this directory – or a file that is *bind mounted* to one of these files – returns a file handle for the corresponding namespace of the process specified by `PID`. As long as this file descriptor remains open, the namespace will remain alive, even if all processes in the namespace terminate. This is the file descriptor that may be passed to `setns(2)`.

2.1.1.2 Types of Namespaces

Next, we briefly discuss all different types of namespaces in the order in which their implementation was completed [55].

- **Mount namespaces**
  Mount namespaces, which were completed by Linux 2.4.19 and correspond to the `CLONE_NEWNS` flag, isolate the set of filesystem mount points seen by a group of processes. Thus, processes in different mount namespaces can have different views of the filesystem hierarchy.

  One use of mount namespaces is to create environments that are similar to `chroot` jails [56, 57]. However, by contrast with the use of the `chroot(2)` system call, mount namespaces are a more secure and flexible tool for this task. Other more sophisticated uses of mount namespaces are also possible; for example, separate mount namespaces can be set up in a master-slave relationship, so that the mount events are automatically *propagated*[58] from one namespace to another.

- **UTS namespaces**
  UTS namespaces, which were completed by Linux 2.6.19 and correspond to the `CLONE_NEWUTS` flag, isolate two system identifiers, the hostname and the NIS domain. These are set using `sethostname(2)` and `setdomainname(2)` system calls and can be retrieved using the `uname(2)`, `gethostname(2)` and `getdomainname(2)` system calls. In the context of containers, the UTS namespaces feature allows each container to have its own hostname and NIS domain name. This can be useful for initialization and configuration scripts that tailor their actions based on these names [48, 55].

- **IPC namespaces**
  IPC namespaces, which were completed by Linux 2.6.19 and correspond to the `CLONE_NEWIPC` flag, isolate certain interprocess communication (IPC) resources, namely, System V IPC objects and (since Linux 2.6.30) POSIX message queues. The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message

queue filesystem [48].

- **PID namespaces**

  PID namespaces, which were completed by Linux 2.6.24 and correspond to the `CLONE_NEWPID` flag, isolate the process ID number space, in the sence that processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending and resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.

  From the point of view of a particular PID namespace instance, a process has two PIDs: the PID inside the namespace, and the PID outside the namespace on the host system. PID namespaces can be nested: a process will have one PID for each of the layers of the hierarchy starting from the PID namespace in which it resides through to the root PID namespace. A process is aware only of processes contained in its own PID namespace and the namespaces nested below that PID namespace [55].

- **Network namespaces**

  Network namespaces, which were completed by about Linux 2.6.29 and correspond to the `CLONE_NEWNET` flag, isolate the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewalls, the `/proc/net` directory, the `/sys/class/net` directory, port numbers (sockets), and so on. A physical network device can live in exactly one network namespace. A virtual network device ("`veth`") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace [48].

  Network namespaces make containers useful from a networking perspective: each container can have its own virtual network device and its own applications that bind to the per-namespace port number space; suitable routing rules in the host system can direct network packets to the network device associated with a specific container. Thus, for example, it is possible to have multiple containerized web servers on the same host system, with each server bound to port 80 in its (per-container) network namespace [55].

- **User namespaces**

  User namespaces, which were completed by Linux 3.8 and correspond to the `CLONE_NEWUSER` flag, isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and `capabilities`. A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of `0` inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace [60].

  Starting in Linux 3.8, unprivileged processes can create user namespaces, which opened up a raft of interesting new possibilities for applications: since an otherwise unprivileged process can hold root privileges inside its own user namespace, unprivileged applications now have access to functionality that was formerly limited to root [55].

- **Control group namespaces**

  Control group namespaces, or *cgroup* namespaces, virtualize the view of a process's cgroups, as seen via the `proc` pseudo-filesystem. Each cgroup namespace has its own set of cgroup root directories. When a process creates a new cgroup namespace using `clone(2)` or `unshare(2)` with the `CLONE_NEWCGROUP` flag, it enters a new cgroup namespace in which its current cgroups directories become the cgroup root directories of the new namespace (this applies for both versions, 1 and 2, of cgroups). As we have not discussed about control groups yet, some readers may be unfamiliar with these concepts and terms, but all these should be clearer after going through the content of the following subsection.

2.1.2 Linux Control Groups

The control groups mechanism, usually referred to simply as "*cgroups*", are a Linux kernel feature which allows processes to be organized into hierarchical groups. The usage of various types of resources on behalf of these groups can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called *cgroupfs*. Grouping is implemented in the core cgroup kernel code, while resource

tracking and limits are implemented in a set of per-resource-type "subsystems" (for memory, CPU, etc).

We refer to a collection of processes which are bound to a set of limits or parameters defined via the cgroup filesystem as a *cgroup*. A *subsystem* is a kernel component that modifies the behavior of the processes that belong in a certain cgroup. There are various subsystems, which thereby provide a variety of functionality, such as limiting the amount of CPU time and memory available to a cgroup, managing the access to the system's devices for the processes in a cgroup, accounting for the CPU time used by a cgroup, freezing and resuming execution of the processes in a cgroup, and many more. Sometimes, they may also be referred to as *resource controllers*, or simply *controllers*[1].

The cgroups for a resource controller are arranged in a *hierarchy*. Hierarchy is a central concept throughout the cgroups mechanism, and is defined by creating, removing and renaming subdirectories within the cgroup filesystem. At each level of the hierarchy, attributes, such as limits, may be defined. In general, the limits, control and accounting provided by cgroups have effect throughout the whole sub-hierarchy underneath the cgroup where the attributes are defined. Therefore, for instance, the limits that may have been placed on a cgroup at a higher level in the hierarchy cannot be exceeded by any of its descendant cgroups.

Although cgroups were initially released in Linux 2.6.24, many of the existing controllers that allow the management of various types of resources have been added later. However, many inconsistencies arose due to the fact that the development of these controllers was largely uncoordinated. As a result, the management of the cgroup hierarchies became rather complex. To mitigate the problems in the initial implementation of cgroups (cgroups version 1), work for a new cgroups implementation had begun since Linux 3.10, the cgroups version 2, which was eventually made official with the release of Linux 4.5. The new version is supposed to replace the old one, however, the latter continues to exist and is unlikely to be removed for reasons related to compatibility. In addition, as the new version only implements a subset of the resource controllers available in the older version, the latter still needs to be used for certain resources; the only restriction is that a controller cannot be employed in hierarchies

[1]Not to be confused with Kubernetes' *controllers*, which are used and will be discussed throughout the rest of the thesis.

of both versions simultaneously.

Rather than delving into the details of each version of cgroups implementation, or into technical details about their usage, we present various available controllers in cgroups v1, to let any reader who is unfamiliar with cgroups better comprehend their purpose, also noting which of them are present in cgroups v2 as well.

- **cpu**
  Through this controller, which is present since Linux 2.6.24, cgroups are guaranteed a minimum number of "CPU shares" when the system is busy. In Linux 3.2 this controller was extended to provide "CPU bandwidth" control, in the sense that it became possible to define an upper limit on the CPU time allocated within a scheduling period to the processes in a cgroup, which applies even if there is no other competition for the CPU. Since Linux 4.15 there has been a successor implementation of the controller in cgroups v2.

- **cpuacct**
  This controller, which is present since Linux 2.6.24, provides accounting for CPU usage by groups of processes. Since Linux 4.15 the functionality of this controller has been merged with the successor implementation of the `cpu` controller, in cgroups v2.

- **cpuset**
  This cgroup, which is present since Linux 2.6.24, can be used to bind the processes in a cgroup to a specified set of CPUs and NUMA nodes.

- **memory**
  The `memory` controller, which is present since Linux 2.6.25, supports reporting and limiting of process memory, kernel memory, and swap used by cgroups. Since Linux 4.5 there has been a successor implementation of the controller in cgroups v2.

- **devices**
  The `devices` controller, which is present since Linux 2.6.26, supports controlling which processes may create devices (`mknod(2)` system call), as well as open

them for reading or writing. The policies are specified via whitelists and blacklists, in which hierarchy is enforced, so new rules must not violate existing rules for the target or ancestor cgroups.

- **freezer**
  The `freezer` controller, which is present since Linux 2.6.28, can be used to suspend and restore (resume) all processes in a cgroup. Freezing a cgroup /A also causes its children, for example, processes in /A/B, to be frozen.

- **net_cls**
  This controller, which is present since Linux 2.6.29, places a `classid`, specified for the cgroup, on network packets created by a cgroup. These `classid`s can then be used in firewall rules or in traffic control tools.This applies only to packets leaving the cgroup, not to traffic arriving at the cgroup.

- **blkio**
  The `blkio` controller, which is present since Linux 2.6.33, provides control and limiting of access to specified block devices by applying IO control in the form of throttling and upper limits against leaf nodes and intermediate nodes in the storage hierarchy. Two policies are available: a proportional-weight time-based division of disk implemented with CFQ[2], which is in effect for leaf nodes using CFQ, and a throttling policy which specifies upper I/O rate limits on a device. Since Linux 4.5 there has been a successor implementation of the `blkio` controller, named **io** controller, in cgroups v2.

- **perf_event**
  This controller, which is present since Linux 2.6.39, allows perf monitoring of the set of processes grouped in a cgroup. Since Linux 4.11 there has been a successor implementation of the controller in cgroups v2.

- **net_prio**
  This controller, which is present since Linux 3.3, allows priorities to be specified for cgroups, per network interface.

- **hugetlb**

[2]Complete Fairness Queueing [62].

This controller, which is present since Linux 3.5, supports limiting the use of huge pages[3] by cgroups.

- **pids**
  This controller, which is present since Linux 4.3, permits limiting the number of process that may be created in a cgroup (and its descendants). Since Linux 4.5 there has been a successor implementation of the controller in cgroups v2.

- **rdma**
  The `rdma` controller, which is present since Linux 4.11, permits limiting the use of RDMA[4]/IB[5]-specific resources per cgroup. Since Linux 4.11 there has been a successor implementation of the controller in cgroups v2.

2.1.3 Union Filesystems

Traditionally, when a filesystem is mounted on a directory, the existing contents of the directory are "hidden", and the content of the most recently mounted filesystem is shown. These hidden files and directories become available again only after the mounted filesystem is unmounted; until then, they are inaccessible to the user, even though they still exist. In the kernel, the filesystems are stacked in order of their mount sequence: the first mounted filesystem is at the bottom of the mount stack, and the latest mount is at the top of the stack. Only the files and directories of the top of the mount stack are visible to the user [66].

Unification filesystems, also known as *union filesystems*, overcome this by providing access to all directories and files present in the directory, even after a mount. With union filesystems, directory entries from the filesystems that are lower in the stack are merged with the directory entries of the filesystems that are higher in the stack, thus making a logical combination of all mounted filesystems, a "union" in the mathematical sense

[3] *Huge pages* are a mechanism that enables the Linux kernel to support memory pages bigger than the standard 4 KiB or 16 KiB size, so that the CPU/OS have less page entries to look-up when they need to. Similar mechanisms exist for other operating systems as well [63].

[4] *Remote Direct Memory Access* is a direct memory access from the memory of one computer into that of another without involving either one's operating system, which permits high-throughput, low-latency networking, and is especially useful in massively parallel computer clusters [64].

[5] *InfiniBand* is a computer-networking communications standard used in high-performance computing that features very high throughput and very low latency. It is used for data interconnect both among and within computers. InfiniBand is also used as either a direct or switched interconnect between servers and storage systems, as well as an interconnect between storage systems [65].

– as in set theory. Files with the same name in a lower filesystem are hidden, as the "higher" ones take precedence. Union filesystems effectively combine the namespaces of two or more file systems together to produce a single merged namespace; they appear to merge the contents of several directories while keeping their physical content separate.

In general, most union filesystems share some basic concepts. The various filesystems that are "merged" together are usually called *branches*. Branch access policies can be read-only, writable, or more complex variations that depend on the permissions of other branches. As we already mentioned, branches are ordered or stacked; more often than not, the branch "on top" is the writable branch and the branch "on the bottom" is read-only. Depending on the implementation, branches can sometimes be re-ordered, removed, added, or their permissions changed on the fly.

A commonly required feature is that when a particular directory entry is deleted from a writable branch, that directory entry should never appear again, even if it appears in a lower branch. Usually this is implemented through a combination of *whiteouts* and *opaque directories*. A whiteout is a directory entry that covers up all entries of a particular name from lower branches. An opaque directory does not allow any of the namespace from the lower branches to show through from that point downwards in the namespace [67].

Various sorts of union filesystems have been around since at least the Translucent File Service (or File System), written around 1988 for SunOS. BSD has had union mounts since 4.4 BSD-Lite, around 1994, and Plan 9 implemented union directories in a similar time frame. The first prototype of a union-style file system for Linux was the Inheriting File System (IFS), written for Linux 0.99 in 1993, but was abandoned in 1998. Later implementations for Linux include `unionfs`[68] in 2003, `aufs`[69] in 2006, and `union mounts`[66] in 2004, as well as various FUSE prototypes and implementations [67]. The latest implementation, which is also part of the mainline Linux kernel since 2014, is `overlayfs`[70].

As various implementations may be different, going through all of them is out of scope of this thesis. In the rest of the subsection, we are discussing about a few more details that concern only OverlayFS – as presented in [70] – since this is the implementation

that is going to be used in our environment via Docker, and has been included in mainline Linux kernel since version 3.18.

OverlayFS combines two filesystems – an "upper" filesystem and a "lower" filesystem. When a name exists in both filesystems, the object in the "upper" filesystem is visible while the object in the "lower" filesystem is either hidden or, in the case of directories, merged with the "upper" object. To be pedantic, it would be more accurate to refer to an upper and lower "directory tree" rather than "filesystem" as it is quite possible for both directory trees to be in the same filesystem and there is no requirement that the root of a filesystem is used for either upper or lower.

The lower filesystem can be any filesystem supported by Linux and does not need to be writable; it can even be another overlayfs. The upper filesystem will normally be writable; if it is, it must support the creation of certain extended attributes, so not all filesystems are suitable. A read-only overlay of two read-only filesystems may use any filesystem type.

Overlaying mainly involves directories. If some name appears in both upper and lower filesystems and refers to a non-directory in either of them, then the lower object is hidden and the name ends up referring only to the upper object. If both upper and lower objects are directories, a directory is formed by merging them. Lower and upper directories are defined by giving them as the mount options `lowerdir` and `upperdir`, and they are merged into another directory, that path of which is also given as an argument. Another required mount option is the `workdir`, which needs to be an empty directory on the same filesystem as `upperdir`, and is used to prepare files before they are switched to the overlay destination in a single atomic action [71].

Since Linux kernel 4.0, overlayfs has been extended to enable multiple lower directories to be specified in the `lowerdir` mount option (delimited by colons), with the rightmost lower directory being on the bottom of the union and the leftmost directory on the top. In this extended version, the `upperdir` is optional, and if it is omitted, then the `workdir` option is also optional and is ignored if provided. In this scenario, the overlay will be read-only.

Every time a lookup is requested in the merged directory, the lookup is performed in each actual directory and the combined result is cached in the dentry belonging to the overlay filesystem. If both actual lookups find directories, both are stored and a

merged directory is created; otherwise only one is stored: the upper if it exists, else the lower. Only the lists of names from directories are merged; other content, such as metadata and extended attributes, are reported for the upper directory only. These attributes of the lower directory remain hidden.

In order to support the deletion of objects without changing the lower filesystem, an overlay filesystem needs to record in the upper filesystem that files have been removed. As we mentioned earlier, this is done using whiteouts and opaque directories. A whiteout is created in the form of a character device with `0:0` major and minor device numbers. When a whiteout is found in the upper level of a merged directory, any matching name in the lower level is ignored, and the whiteout itself is also hidden. A directory is made opaque by setting a properly setting a corresponding extended attribute. In the case of opaque directories in the upper filesystem, any directories in the lower filesystem with the same names are ignored.

Lower layers may be shared among several overlay mounts and that is indeed a very common practice. An overlay mount may use the same lower layer path as another overlay mount and it may use a lower layer path that is beneath or above the path of another overlay lower layer path. Using an upper layer path or a `workdir` path that are already used by another overlay mount is not allowed. If files are accessed from two overlayfs mounts which share or overlap the upper layer or `workdir` path the behavior of the overlay is undefined, though it does not result in a crash or a deadlock. Mounting an overlay using an upper layer path, where the upper layer path was previously used by another mounted overlay in combination with a different lower layer path, is allowed only under certain circumstances.

Offline changes, when the overlay is not mounted, are allowed to either the upper or the lower trees. Changes to the underlying filesystems while part of a mounted overlay filesystem are not allowed. If the underlying filesystem is changed, the behavior of the overlay is undefined, though it does not result in a crash or a deadlock.

Union file systems are particularly appealing because, among other reasons, they enable an interesting pattern. Read-only "images" can be augmented with a writable layer, thereby enabling dynamic, ephemeral sessions. Effectively, this is COW[6]: a resource-management technique in which read-only data that need to be duplicated

[6]Copy-On-Write.

defer the actual copy operation until they need to be modified for the first time, whereupon they are copied and modified in a read-write layer. The COW mechanism is popular among container runtime environments for creating filesystems for Linux containers. Even though it is not the only option for assembling container filesystems, it is one of the more performant because it allows pages in the kernel's page cache to be shared between containers.

2.1.4 Docker

In the words of its backing company, Docker Inc., Docker is an open platform for developing, shipping, and running applications which are separated from the underlying infrastructure, thus enabling easy and quick software delivery and infrastructure management. This is achieved, basically, by providing the ability to package and run applications in containers via the *Docker Engine*, its container runtime engine. In addition, Docker provides tooling and a platform to manage the lifecycle of the containers, aiming to aid the development of the applications and their supporting components using containers, render the container the unit for distributing and testing these applications, and finally deploy them into production environments, regardless of whether that is a local datacenter, a cloud provider or a hybrid of the two.

In order for a reader quickly grasp its purpose, deployments using Docker are often juxtaposed with deployments using traditional virtual machines, as shown in Figure 2.1. Using traditional virtualization technology, a virtual machine runs a full-blown "guest" operating system with virtual access to host resources through a *hypervisor*, often providing an environment with more resources than what most applications actually need. By contrast, using operating-system-level virtualization containers run natively on Linux and share the kernel of the host machine with other containers. Containers run discrete processes, taking no more memory than any other executable, thus such deployments are much more lightweight than deployments using virtual machines.

Docker consists of a Community Edition and an Enterprise Edition, Docker CE and Docker EE respectively. Nonetheless, a large portion of the codebase is open source, released under the terms of the Apache 2.0 license.

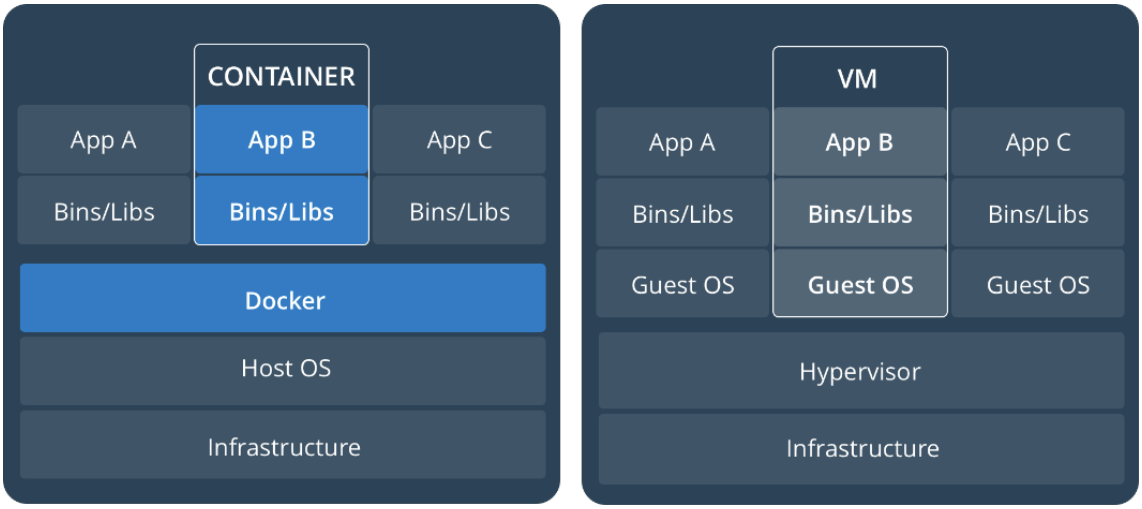


Figure 2.1: *Deployments based on operating-system-level virtualization using Docker containers juxtaposed with deployments based on traditional virtualization technology using a hypervisor and virtual machines. [72]*

2.1.4.1 Docker Engine

The *Docker Engine* is a client-server application with three major components, which are illustrated in Figure 2.2:

1. a server process running as a daemon, which is simply called Docker daemon or `dockerd`;
2. a REST API to allow third-party applications to communicate with Docker daemon and instruct it what to do;
3. a CLI[7] application that acts as a client to Docker daemon, and is known as the `docker` command.

In Docker's client-server architecture, which is shown in Figure 2.3, the Docker client talks through the REST API to the Docker daemon. These two entities may run on the same host machine or on different ones, thus the communication takes place over either Unix sockets or a network interface. The Docker deamon, essentially, is responsible for building, running and distributing the containers. It listens for Docker API requests and manages Docker "objects", which include images, containers, networks, volumes and plugins.

[7]Command Line Interface.

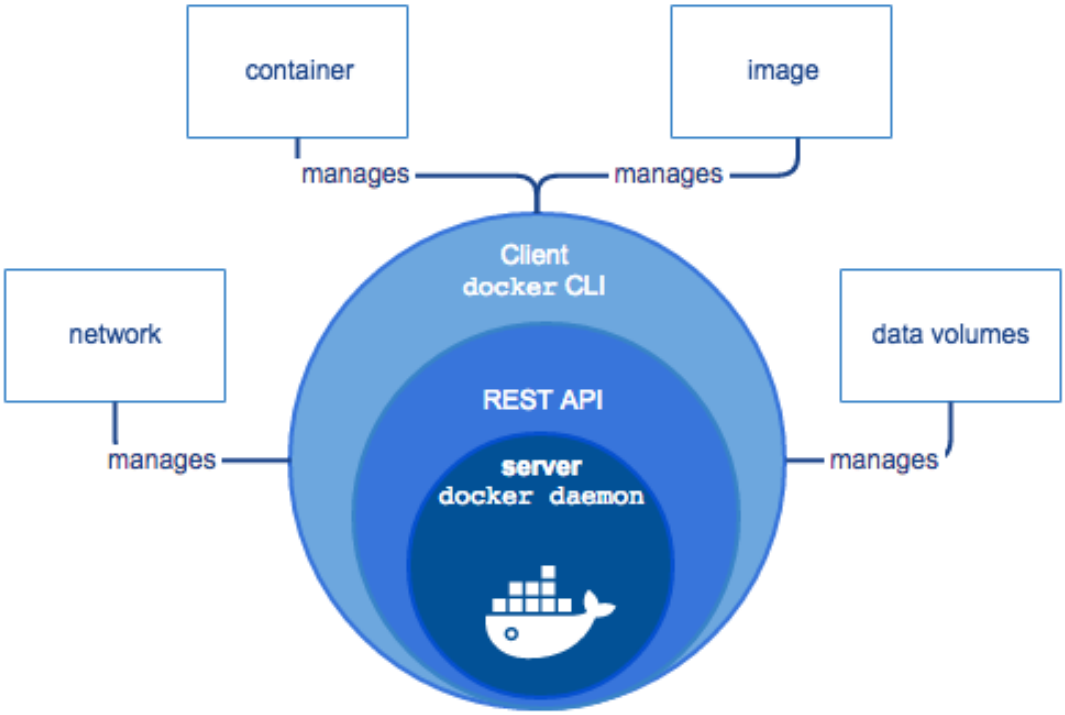


Figure 2.2: *High level illustration of the components of Docker Engine. Docker CLI uses the REST API to control or interact with the Docker daemon through either scripting or direct CLI commands. Docker daemon creates and manages Docker "objects", including images, containers, networks, and volumes. [73]*

In Figure 2.3 we can observe another component in the high level architecture, other than the Docker host and the Docker client, which may or may not run on the same host machine. This component is the Docker *registry*. A Docker registry is merely a repository for storing Docker images, which are going to be discussed soon. There are registries that are publicly available, such as the Docker Hub, which has been used in our development process. However, it is also possible for anyone to run their own private Docker registries, to organize the stored images as deemed best for each set of use cases.

2.1.4.2 Images & Containers

In the world of Docker, images and containers are two different things, yet closely related. An image is an inert, **immutable** file that's essentially a snapshot of a container. It is built up from a series of layers, which – in the case of Docker – may represent

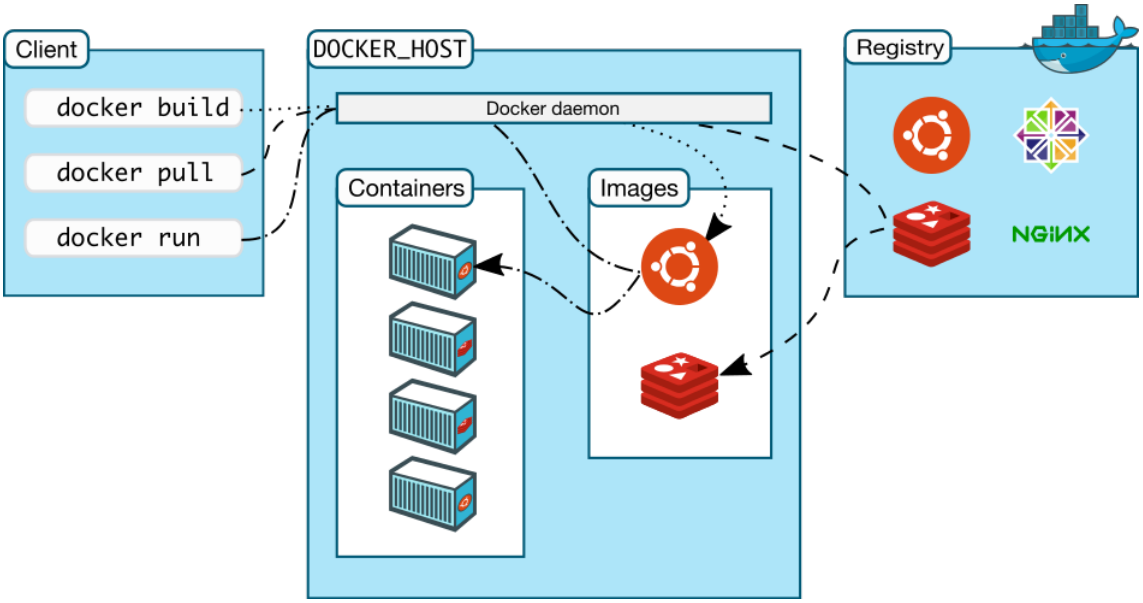


Figure 2.3: *High level architecture of the Docker Engine. [73]*

instructions given in a "*Dockerfile*". The layers are read-only, and each layer is merely a set of differences from the layer before it, as they are stacked on top of each other.

To create a new container, Docker adds a new writable layer on top of the underlying layers of an image. This layer is also called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. Docker calls *storage driver* the mechanism that handles the details about the exact manner that all these layers interact with one another. Various storage drivers are available, and some of them are based on union filesystems, which were examined briefly in subsection 2.1.3; specifically, there is one such storage driver based on AUFS and two storage drivers based on OverlayFS.

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted whereas the underlying image remains unchanged. Using a programming metaphor, if an image is a class, then a container is an instance of a class – a runtime object. Figure 2.4 shows a container which is based on the `ubuntu:15.04` image; image layers are read-only and the container layer on top of them is read-write.

When images need to be downloaded from a registry, each layer is downloaded separately. Then, it is cached locally by Docker daemon for future use: in the case that two

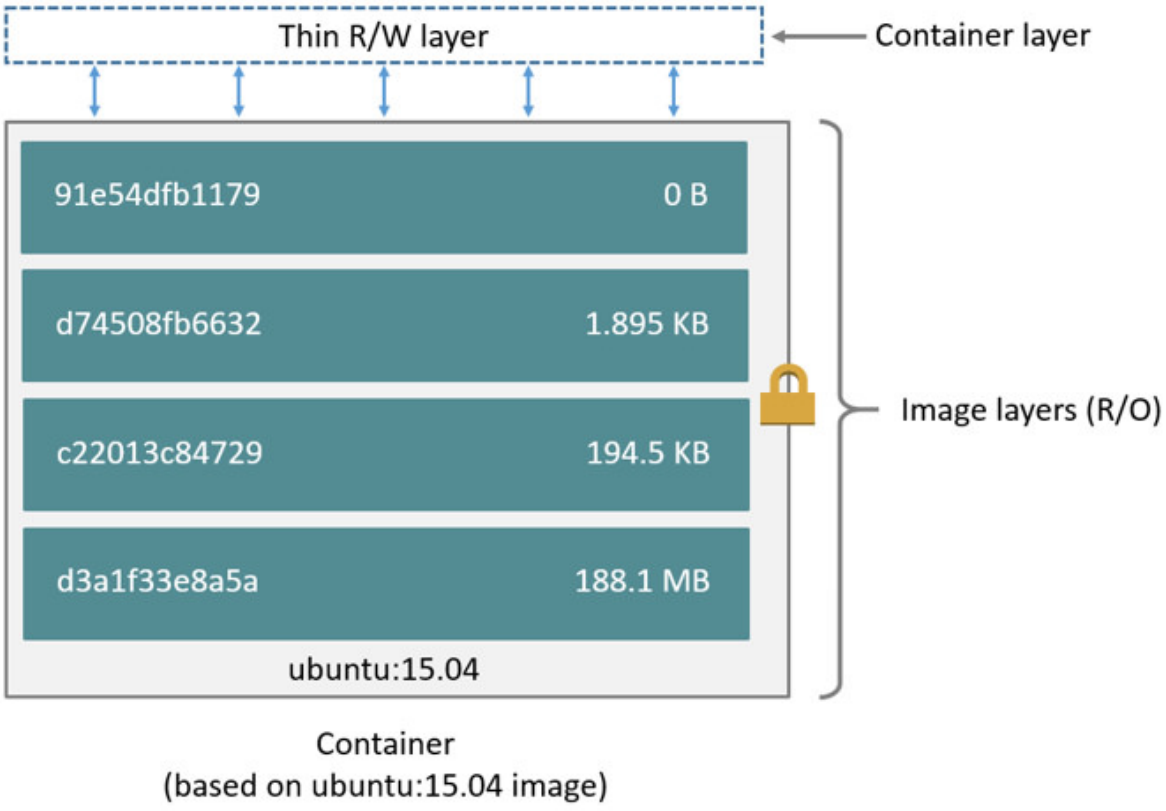


Figure 2.4: *A container which is based on the* `ubuntu:15.04` *image. Image layers are read-only and the container layer on top of them is read-write. [12]*

or more images that must be used are comprised of some common layers, those layers will not be needed to be downloaded multiple times.

Since each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. This is effectively a Copy-On-Write mechanism in action: if a file or directory exists in a lower layer within the image, and another layer – including the writable layer – needs read access to it, it just uses the existing file. On the first time that another layer needs to modify the file, either while building the image or while running the container, the file is copied into that layer and that copy is modified.

This behavior minimizes both I/O and the size of each of the subsequent layers. Any files the container does not change do not get copied to this writable layer at all, which results in the writable layer being as small as possible. Not only does Copy-On-Write save space, but it also reduces the start-up times of containers: when one or multiple

containers need to be spawned from an image – possibly the same image – Docker only needs to create the thin writable container layer. If instead it had to make an entire copy of the underlying image stack each time it started a new container, container start times and disk space used would be significantly increased.

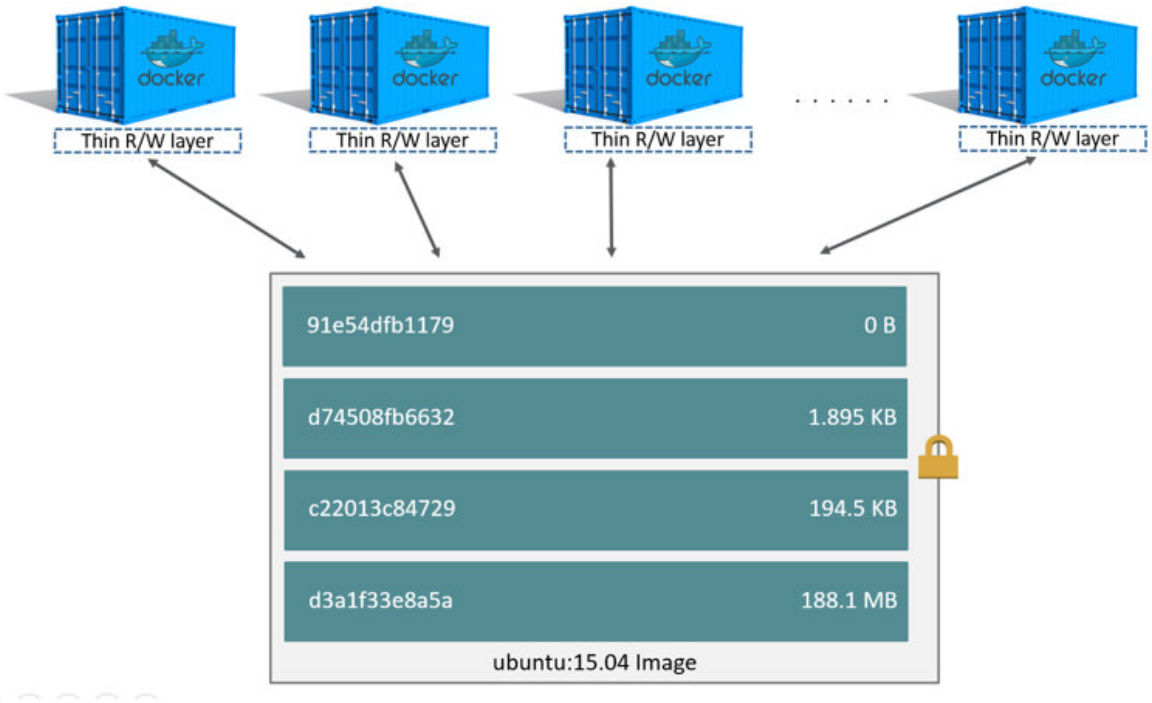


Figure 2.5: *An image, again based on the* `ubuntu:15.04` *image, which is related to multiple spawned containers. They all share the read-only layers of the common image, and only the thin writable container layer is different for each of them. [12]*

Figure 2.5 illustrates an example of an image, again based on the `ubuntu:15.04` image, which is related to multiple spawned containers. They all share the read-only layers of the common image, and only the thin writable container layer is different for each of them.

2.2 Kubernetes

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services. It aims to facilitate both declarative configuration and automation, and it has a large, rapidly growing ecosystem. Kubernetes provides a container-centric management environment. It orchestrates computing, networking,

and storage infrastructure on behalf of user workloads. This provides much of the simplicity of PaaS[8] with the flexibility of IaaS[9], and enables portability across infrastructure providers.

It is noted that, unless explicitly stated or cited otherwise, the vast majority of the information regarding Kubernetes and its rationale that are presented in this section, originate either from publications by Google [75, 76, 77], or from Kubernetes' documentation pages [78, 79, 80, 81, 82, 83, 84, 85, 86].

2.2.1 Precursors

Engineers at Google have been using custom containerization-like methods for deploying applications and systems on their datacenters long before Docker became popular [75]. In fact, later they were actively involved in the development of the cgroups mechanism in the Linux kernel [74]. The first "container management" system that was developed and used internally at Google was Borg[76]. Built to manage both long-running services and batch jobs, it shares machines between these two types of applications as a way of increasing resource utilization and thereby reducing costs. As more and more applications, with diversified needs, were developed to run on top of Borg, a broad ecosystem of tools and services for it was developed. The ecosystem included systems that provide mechanisms for configuring and updating jobs, predicting resource requirements, dynamically pushing configuration files to running jobs, service discovery and load balancing, auto-scaling, machine-lifecycle management, quota management, and many more.

In an effort to improve the software engineering of the Borg ecosystem, Omega[77] was born. It was built from the ground up in order to have a more consistent and principled architecture, but it applied many of the patterns that had been proven successful in Borg. An important architectural difference compared to Borg is that Omega stored the state of the cluster in a centralized Paxos-based[119] transaction-oriented store that was accessed by the different parts of the cluster control plane (such as schedulers), using *optimistic concurrency control* to handle the occasional conflicts. This

[8] Platform as a Service.
[9] Infrastructure as a Service.

decoupling was crucial, because it allowed Omega's functionality to span over multiple architectural components that acted as peers, rather than relying on a monolithic, centralized master component that enfolds all of the system's functionality, almost heralding the upcoming spread of Microservices Architecture.

2.2.2 Conception

The latest container management system that is conceived and developed at Google is Kubernetes, which, in contrast to Borg and Omega, is an open source project released under the terms of the Apache License (Version 2.0), and it is heavily inspired by those two systems.

The conception of Kubernetes is due to the realization that the benefits of containerization go beyond the technical aspects of containers themselves as an operating-system-level virtualization implementation. Containerization transforms the datacenter from being machine-oriented to being application-oriented, by abstracting away various details related to machines and operating systems from the application environment and the deployment infrastructure. It also shifts the management APIs from machine-oriented to application-oriented, which is in alignment with the "pet versus cattle" analogy[34] for machines in the datacenter, and was mentioned earlier.

This shift brings in many benefits. Obviously, it relieves both the application developers and the operations teams from worrying about specific details of the machines and the operating systems. Consequently, it provides the infrastructure teams flexibility to roll out either new hardware or operating system upgrades, while minimizing the impact that these may have on the applications currently deployed, thus on their developers, too. In addition, it ties telemetry collected by the management system to applications rather than machines, which dramatically improves application monitoring and introspection, especially when scale-up, machine failures, or maintenance cause application instances to move. In the other direction, the container management system can communicate information into the containers, such as resource limits, container metadata for propagation to logging and monitoring (e.g. user name, job name, identity), and notices that provide graceful termination warnings, e.g. in advance of node maintenance [75].

2.2.3 API & Object Model

2.2.3.1 Containers & Pods

Although we have been discussing about containerization, so far we have refrained from using the term "container-centric", in favor of the more generic term "application-centric", to describe the transformation that has happened. Indeed, applications do not need to be mapped onto containers bijectively (i.e. there is no need to have exactly one container per application). In reality, it is often useful to use some sort of "nested containers": a simple two-layer scheme where the outermost layer provides a pool of resources while the inner ones provide further deployment isolation. Technically, this is nothing more than some of the containers running on a host merely sharing some of the resources provided by the isolation mechanism; for instance, they could "live" in some common Linux namespaces, thus having a shared view of some of their underlying resources, such as storage and networking.

As a matter of a fact, Kubernetes enforces containers to always run under such a nested scheme, which is defined via "`Pods`" and "`Containers`". A `Pod` corresponds to the outermost layer of the scheme, and a number of `Containers` may be deployed inside it. Except from one or more application containers, a `Pod` also encapsulates storage resources, a unique network IP, and options that govern how the containers should run. Most importantly, it represents a *unit of deployment*: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources. Docker is the most common container runtime used, but other container runtimes are supported as well [78].

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated. Each Pod is assigned a unique IP address and containers in a Pod share the same network namespace. Therefore, containers inside a Pod can communicate with one another using `localhost`, and when containers in a Pod communicate with entities outside the Pod, they must coordinate how they use

the shared network resources (such as ports). Furthermore, all containers in the Pod can access a set of shared volumes, thus are allowed to share data. Certain types of volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted.

Each Pod is meant to run a single instance of a given application. To scale an application horizontally, multiple Pods should be used rather than multiple containers within the same Pod. In Kubernetes, this is generally referred to as *replication*. Replicated Pods are usually created and managed as a group by an abstraction called a "*Controller*". There are several types of Controllers, and some of them will be discussed later in this section. Individual Pods are seldom created directly in Kubernetes. Pods are designed as ephemeral, disposable entities. Upon its creation, either directly or indirectly (through a Controller), a Pod is scheduled to run on a Node of the Kubernetes cluster. It stays there until all `Containers`' processes are terminated, the `Pod` API object is deleted, the Pod is evicted due to lack of underdlying resources, or the Node fails [78].

Despite one of Kubernetes' major features being the support for self-healing deployments, Pods do not self-heal by themselves. A Pod will not survive in any of the conditions cited above. Handling the work of managing the ephemeral, disposable Pod instances is a job for the higher-level abstraction, the Controller. A Controller can create and manage multiple Pods, handling replication and rollout and providing the self-healing capabilities at cluster scope. For instance, on a Node failure, the Controller might automatically replace the Pod by scheduling an identical replacement on a different Node. A nonextensive list of examples of such Controllers would include the `Deployment`, the `StatefulSet`, the `DaemonSet`, and the `Job`.

2.2.3.2 Controllers

Kubernetes Controllers constitute a very interesting topic with respect to the subject of this thesis. Nevertheless, it is a vast topic that might also include extensive commentary on design decisions made during the development of Kubernetes, as well as best practices regarding the usage of Controllers. Rather than initiating such a detailed discussion, we selectively merely present a quick overview of some of the relevant information, lest we digress a lot from the original purpose of the thesis.

Before overviewing the functionality of some types of Controllers, we have to emphasize on a couple of common concepts behind all of them, as well as behind Kubernetes' rationale in general: the *control loop* and the *labels*.

API consistency & the control loop (or reconciliation loop)

Every kind of API object in Kubernetes has three basic fields in its description: the `ObjectMetadata`, the `Specification` or `Spec`, and the `Status`. The former, the `ObjectMetadata` has similar content for all objects in the system: it contains information such as the object's name and unique identifier (UID), its object version number (useful for the optimistic concurrency control) and labels (which will be discussed right next). The content of `Spec` and `Status` fields varies, depending on the exact type of the API object. Nonetheless, their concept remains always the same: `Spec` describes the *desired state* of the API object, while `Status` contains read-only information regarding its *current state*.

The evident uniformity of Kubernetes' API provides, by itself, many benefits. For instance, the system and its API becomes easier to learn, use, and extend via custom generic tools which may work for all kinds of API objects. Most importantly though, it allows the API to be consistent, in the sense that the same API is used by all people, internal Kubernetes components and external automation tools.

To further this consistency, the vast functionality of the cluster is distributed among multiple components, and the functionality of each of these components is decoupled from the functionality of the rest of them. The separation of concerns between API components means that higher-level services can all share the same common basic building blocks. So far, we have discussed about Pods, which provide an abstraction to specify the containers that need to be run. As we will discuss soon, these are the basic building blocks for other API components of the system, which, in turn, may also be used as building blocks for even higher-level API abstractions, under certain circumstances.

The aforementioned consistency enables a powerful design pattern, which is common for many Kubernetes components and aims to improve the resilience of the system overall: the *control loop* or *reconciliation loop* or simply *controller pattern*. The idea behind it is simple, and stems from concepts of control theory: the *desired state* of an

object is continually compared against its observed *current state*, and actions are taken (on behalf of the appropriate component) to converge the two states. Reconciliation loops are robust to failures and perturbations in the system because in such cases they allow it to pick up where it left off.

Labels & Selectors [79]
First of all, `labels` are key-value pairs that are attached to Kubernetes API objects, such as Pods. `Labels` are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. They can be used to organize and to select subsets of objects. `Labels` can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key-value `labels` defined, and each key must be unique for a given object.

`Labels` enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings. By design, they do not aim to provide uniqueness. Quite the contrary, in general, we expect many objects to carry the same `labels`. Via a `label selector`, the user is capable of identifying a set of API objects. The `label selector` is the core grouping primitive in Kubernetes.

The API currently supports two types of `selectors`. *Equality-based* and *inequality-based* requirements allow filtering by `label` keys and values. Matching API objects must satisfy all of the specified `label` constraints, though they may have additional `labels` as well. *Set-based* `label` requirements allow filtering keys according to a set of values. A `label selector` can be made of multiple requirements which are comma-separated. In the case of multiple requirements, all must be satisfied; hence, the comma separator acts as a logical `AND` operator. `List` and `Watch` API operations may also specify `label selectors` to filter the sets of objects returned using a query parameter.

ReplicationController & ReplicaSet [80, 81]
`ReplicationControllers`, and their successors, `ReplicaSets`, ensure that a specified number of Pod replicas are always running at any one time. In other words,

ReplicationControllers and ReplicaSets make sure that a Pod or a homogeneous set of Pods is always up and available. The set of Pods is tracked using the labels and label selectors mechanism that was described above.

ReplicationControllers and ReplicaSets simply ensure that a – pre-specified – desired number of Pods matches its label selector and are operational. If there are too many Pods, the ReplicationController or ReplicaSet terminates the extra Pods. If there are too few, the ReplicationController or ReplicaSet starts more Pods. Unlike manually created Pods, the Pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, Pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, using a ReplicationController or a ReplicaSet is recommended, even if an application requires only a single Pod. All in all, ReplicationControllers and ReplicaSets are similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

The only difference between a ReplicationController and a ReplicaSet, as of August 2018, is that the latter, being the former's successor implementation, supports API objects' filtering using *set-based* label selectors.

Deployment [82]

A Deployment controller provides declarative updates for Pods and ReplicaSets. A desired state is described in a Deployment API object, and the Deployment controller changes the actual state to the desired state at a controlled rate. New ReplicaSets may be created by defining Deployments, or existing Deployments can be removed, and have all their resources adopted by new Deployments. ReplicaSets owned by a Deployment are not intended to be managed directly by the user. Every possible use case should be covered by manipulating just the Deployment object.

ReplicaSet, being the next-generation ReplicationController is used directly by Deployment as a mechanism to orchestrate Pod creation, deletion and updates. Although all ReplicationController, ReplicaSet and Deployment are top-level Kubernetes API objects, Deployment is almost always the recommended approach to deploy and operate stateless applications over Kubernetes – with the exception being

cases of custom update orchestration requirements.

StatefulSet [83]
A `StatefulSet` controller manages the deployment and scaling of a set of Pods, and provides guarantees about the *ordering* and *uniqueness* of these Pods. The selection of Pods of that set takes place, again, using the `label selector` mechanism. Like a `Deployment`, a `StatefulSet` manages Pods that are based on an identical container specification. Unlike a `Deployment`, a `StatefulSet` maintains a sticky identity for each of their Pods. These Pods are created from the same specification, but are not interchangeable: each has a persistent identifier that is maintained across any rescheduling.

A `StatefulSet` operates under the same pattern as any other Controller. A desired state is defined through a (top-level) Kubernetes API object, called also `StatefulSet`, and the `StatefulSet` controller makes all necessary updates to reach it from the current state. In general, `StatefulSets` are usually valuable for applications that require one or more of the following:

- unique network identifiers, persistent across Pod rescheduling;
- persistent storage across Pod rescheduling;
- ordered, graceful deployment and scaling;
- ordered, graceful deletion and termination;
- ordered, automated rolling updates.

If an application does not require any such persistent identifiers or ordered deployment, deletion, or scaling, it should probably be deployed using some other kind of Controller that provides a set of stateless replicas, such as `Deployment`, `ReplicaSet` or `DaemonSet`.

2.2.3.3 Pod Networking

Kubernetes Pods are mortal: they are born and when they die, they are not resurrected. Various Controllers, such as `ReplicaSets`, may create and destroy Pods dynamically

(e.g. when scaling up or down). Therefore, even though each Pod is assigned its own IP address, those IP address cannot be relied upon to be stable over time. This leads to a problem: if some set of Pods provides functionality to other Pods inside the Kubernetes cluster, how do the latter find out and keep track of which of the former are in that set? Essentially, this is the problem of *service discovery*.

A Kubernetes Service[84] is an abstraction that defines a logical set of Pods, as well as a policy by which to access them – it can be thought of as a "microservice". The set of Pods targeted by a Service is usually determined by the `label selector` mechanism, although this is not always the case.

A `Service` in Kubernetes is a top-level REST API object, similar to a Pod. Like all of the REST API objects, a `Service` definition can be `POST`ed to the Kubernetes API server to create a new instance. Its specification includes a name for the Service, which can be used by cluster addons (e.g. KubeDNS) to provide a set of DNS records for discovering it, and pairs of ports, since Services are a transport layer (layer 4 – TCP/UDP over IP) construct, and they only support the TCP and UDP protocols. Each pair of ports signifies a mapping between a Service's port and a Pod's port (referring to one of the Pods that are tracked by this Service).

Each Service is assigned a "virtual" IP address, which is sometimes called "cluster IP". This is the IP address that may be used by Pods, in combination with one of the Service's ports, to communicate with the set of Pods that are being tracked by the Service. Technical details about the specifics of the assignment and the routing and load balancing based on these virtual IP addresses will not be presented here, but, basically, it involves a separate binary, the `kube-proxy`, which always runs on every host in the Kubernetes cluster. These may be implemented in many different ways, using either `netfilter` (e.g. `iptables` or `IPVS`) which can be very efficient, or even round-robin proxying implemented purely on the userspace.

Published Services may be of one of the four types:

- **ClusterIP**

  The Service is exposed in a Kubernetes cluster-internal IP, making the service reachable only from within the cluster.

- **NodePort**
  The Service is exposed on each Node's IP address at a static port, making the service reachable externally by using some Node's IP address and the random port that the Service has been exposed at.

- **LoadBalancer**
  The Service is exposed externally using a cloud provider's load balancer.

- **ExternalName**
  The Service is mapped to the contents of a specified DNS name by returning a `CNAME` record with its value, instead of setting up proxying of any kind.

Alternatively, Services may be "*Headless*", which allows developers to reduce the coupling to Kubernetes by giving them freedom to do discovery in their own way. Headless Services are not assigned a single cluster IP address, and no load balancing or proxying is performed for them – it is not the `kube-proxy` handling them. Such Services are used by `StatefulSets` to control the domain of their Pods.

2.2.4 Architecture

To deliver its extended functionality, Kubernetes' architecture is structured based on multiple components.

On the one hand, there are master components, which provide the Kubernetes cluster's control plane. Master components make global decisions about the cluster, including detecting and responding to cluster events according to the reconciliation loop. Even though they can be run on any machine in the cluster, quite often they all run on the same host, and no other user containers are scheduled on this host.

A high level illustration of the overall architecture of Kubernetes is deplicted in Figure 2.6.

An overview of the core *master components* is presented below:

- **etcd**
  The well known consistent and highly-available key-value store is used as Kubernetes' backing store for all cluster data. This includes every bit of information

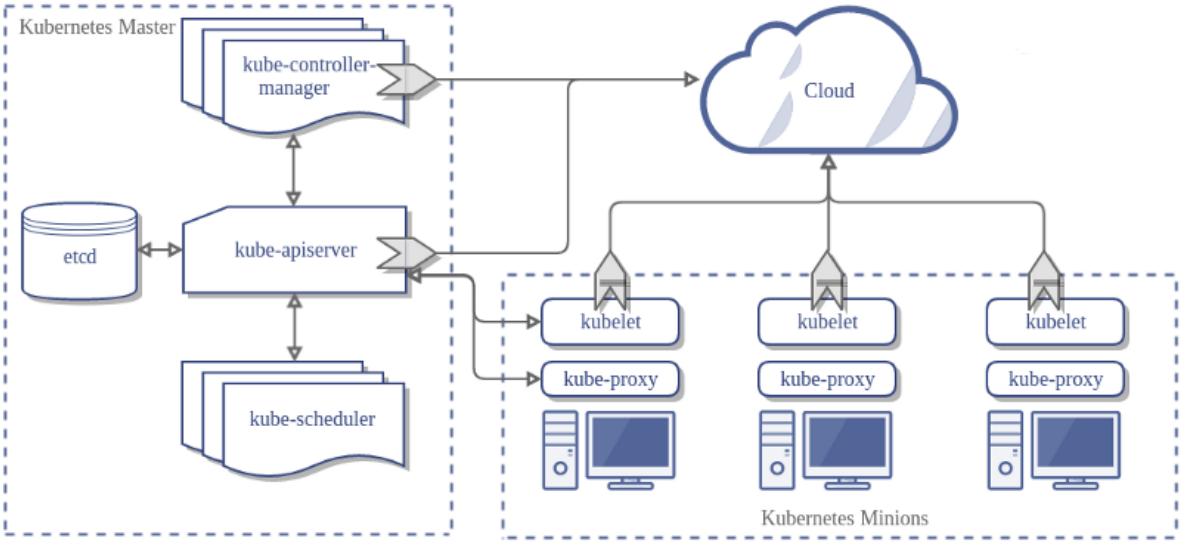


Figure 2.6: *High level architecture of a Kubernetes cluster. [85]*

regarding cluster's state, such as what are the existing Nodes in the Kubernetes cluster, what Pods should be running at any time, which Node should each Pod run on, and much more. Only the API server may connect to etcd and either query or store information in it; all other components must go through the API server to work with the cluster's state.

- **API server**

  It is the master component that is responsible for exposing the Kubernetes REST API. It services all REST operations, thus functioning as the frontend for the Kubernetes control plane. It also validates and configures data for various API objects, and anyone (either component or client) that needs to go retrieve or store information in Kubernetes' backing store needs to go through the API server. It should be noted that the API server is designed to scale horizontally –- i.e. it scales by deploying more instances.

- **scheduler**

  This is a component on the master that watches newly created Pods that have not been bound to a Node yet, and selects a Node for them to run on. Factors taken into account for scheduling decisions include individual and collective resource requirements; hardware, software and policy constraints; affinity and anti-affinity specifications; data locality; inter-workload interference and deadlines. In addition, the scheduler is designed to be pluggable and replaceable by

user-implemented schedulers, and multiple schedulers can run simultaneously on the cluster.

- **Controller manager**
  This is the master component that embeds all Controllers, i.e. all control loops that watch the shared state of the cluster through the API server and make changes attempting to move the currently observed state towards the desired one. Except from the Controllers that were briefly described earlier, there are Controllers for `Nodes`, `Endpoints`, `Services`, `ServiceAccounts`, and many more.

Next, we outline the functionality of the *node components*, i.e. components which run on every node maintaining running Pods and providing the Kubernetes runtime environment.

- **kubelet**
  This is the primary node agent that runs on every node in the Kubernetes cluster. It takes a set of Pod specifications – which may be provided through various mechanisms – and ensures that the containers described in those PodSpecs are running and healthy. Thereby, it is the component that actually communicates with the container runtime, e.g. Docker, of the host, although, of course, it does not manage containers which were not created by Kubernetes.

- **kube-proxy**
  This component runs on every node and enables the Kubernetes Service abstraction that was described earlier. It watches for additions or removals of `Services` and `Endpoints` and properly adjusts the networking rules on the host to correctly perform any connection forwarding.

- **Container runtime**
  This is simply the component that is responsible for running containers. Kubernetes supports several container runtimes: Docker, rkt, runc and any OCI[87] runtime-spec implementation.

2.3 rsync

Rsync is basically a very efficient data synchronization protocol and utility. It has been originally designed and implemented by Andrew Tridgell and Paul Mackerras in 1996, but many people have contributed to it since then, as it was released under the GNU GPLv3 License as free and open source software. It is presented in detail in [88], a dissertation written in 1999.

Rsync can transfer files either to or from a remote host, or locally on one host, but it does not support transferring files between two remote hosts. The synchronization of the data, in particular the files, simply means that both copies of the file are guaranteed to be the same in the end of the procedure. There are two different ways for rsync to contact a remote system: either using a remote-shell program as the transport, such as `ssh` or `rsh`, or contacting an rsync daemon directly via TCP. It offers a large number of options that control many aspects of its behavior and permit very flexible specification of the set of files to be transferred.

Rsync is famous for its delta-transfer algorithm, which reduces the amount of data sent over the network. By default, it finds files that need to be transferred using a "quick check" algorithm that looks for files that have some change either in size or regarding the time of their most recent modification. Any changes in the other file attributed that are being preserved (as requested by the configuration of the transfer) are applied on the destination file directly when the quick check indicates that the file's data does not need to be updated.

All of the information presented in this section originates mostly from [89, 90], but also from [91, 92, 88].

2.3.1 Processes & Roles

When discussing about rsync, specific terms are used to refer to various processes and their roles in the task performed by the utility. Below, we define a few terms that are used in the role and process contexts and will be used hereinafter.

- **client** (*role*)
  The client initiates the synchronization.

- **server** (*role*)
  The remote rsync process or system to which the client connects either within a local transfer, via a remote shell or a network socket. This is a general term and should not be confused with the daemon.

- **daemon** (*role and process*)
  An rsync process that awaits connections from clients. On a certain platform this would be called a service.

- **remote shell** (*role and set of processes*)
  One or more processes that provide connectivity between an rsync client and an rsync server on a remote system.

- **sender** (*role and process*)
  The rsync process that has access to the source files being synchronized.

- **receiver** (*role and process*)
  As a role the receiver is the destination system. As a process the receiver is the process that receives updated data and writes them to the disk.

- **generator** (*process*)
  The generator process identifies changed files and manages the file level logic.

2.3.2 Starting Up

Every time an rsync client is started, first it establishes a connection with a server process. This connection may be through pipes or over a network socket.

In cases that the rsync client communicates with a remote non-daemon server via a remote shell, it starts by forking the remote shell, which then starts an rsync server on the remote system. Both the rsync client and server processes are communicating via pipes through the remote shell. From the point of view of the rsync processes, there is no network at all. In this mode the rsync options for the server process are passed on the command-line that is used to start the remote shell.

When rsync is communicating with a daemon, it is communicating directly with a network socket. In fact, this is the only kind of rsync communication that really is aware of the network. In this mode the rsync options must be sent over the socket.

In the beginning of the communication between the client and the server, each of them sends the maximum protocol version that it supports to the other side. Then, each side uses the minimum value as the the protocol level for the transfer. If this is a daemon-mode connection, rsync options are sent from the client to the server, and then the exclude list is transmitted. From this point on, the relationship between the client and the server is relevant only with regards to error and log message delivery.

Rsync jobs that are local, i.e. ones where the source and destination are both on locally mounted filesystems, are handled exactly like a push. The client, which becomes the sender, forks a server process to also fulfill the receiver role, and the client-sender and server-receiver communicate with each other over pipes.

2.3.3 The File List

The file list includes files' pathnames, but also file metadata regarding the ownership, mode, permissions, size and modification time. If the `--checksum` option has been specified, it also includes calculated checksums of the file.

When the startup procedure that was described above is completed, the sender creates the file list. During its creation, its entries are being transmitted to the receiving side in a network-optimized way. When this finishes, each side sorts the file list lexicographically by path relative to the base directory of the transfer. The exact sorting algorithm that is being used may vary, depending on the protocol version that is in effect. From this point on, any reference to a file in the list is performed using its index in this file list.

When the whole file list has been received by the receiver, it forks off the generator process.

2.3.4 The Transfer

Rsync is heavily pipelined in the sense that it consists of a set of processes that communicate in a – largely – unidirectional way. Once the file list has been shared, the pipeline comprises the generator, the sender and the receiver. The output of the generator is the input of the sender and the output of the sender is the input of the receiver.

Each process runs independently, and may be delayed either when the pipelines stalls, or when it has to wait for disk I/O to be completed or CPU resources to be available.

The generator process compares the file list with its local directory tree. Prior to beginning its primary function, if the `--delete` option has been enabled, the sender identifies local files missing from its own side and makes sure to also delete them on the receiver's side. The generator then goes through the file list, checking each file to figure out whether it can be skipped from the transfer.

By default rsync determines which files differ between the sending and receiving systems by checking the modification time and size of each file. If modification time or size is different between the systems, it transfers the file from the sending to the receiving system. As this only requires reading file directory information, it is quick, but it misses unusual modifications which change neither. Rsync performs a slower but comprehensive check if invoked with the `--checksum` option enabled. This forces a full checksum comparison on every file present on both systems, using the MD5 hash function (or MD4 in older versions of the protocol). Barring rare checksum collisions, this avoids the risk of missing changed files at the cost of reading every file present on both systems. Note that, anyway, rsync always verifies that each transferred file was correctly reconstructed on the receiving side by checking a whole-file checksum that is generated as the file is transferred.

The receiver splits its copy of the file into chunks and computes two checksums for each of them: an MD5 hash, and a weaker but easier to compute "rolling checksum". It transmits these checksums to the sender.

The sender quickly computes the rolling checksum for each chunk in its own version of the file; if they differ, the file must be sent. Otherwise, the sender uses the more computationally expensive MD5 hash to verify that the chunks are the same. It then sends the receiver those parts of its file that did not match, along with information on where to merge these blocks into the receiver's version. This is what makes the copies identical eventually. There is a small probability that differences between chunks in the sender and receiver are not detected, and thus remain uncorrected. With 128 bits from MD5 plus 32 bits from the rolling checksum, the probability is on the order of $2^{-(128+32)} = 2^{-160}$. The rolling checksum used in rsync is based on Mark Adler's adler-32[93] checksum, which is used in `zlib`.

The more sections of the sender's and the receiver's versions are common, the fewer the data required to be trasferred by the utility so that the files end up synchronized. Using typical data compression algorithms, files that are similar when uncompressed may be very different when compressed, and thus the entire file needs to be transferred. Some compression utilities, such as `gzip`, provide a special "rsyncable" mode which allows these files to be efficiently "rsynced", by ensuring that local changes in the uncompressed file yield only local changes in the compressed file.

Rsync supports other key features that aid significantly in data transfers or backup. They include compression and decompression of data block by block using `zlib`, and support for protocols such as `stunnel`.

2.3.5 The Daemon

As it is typical for many Unix daemons, the rsync daemon process is supposed to run all the time, and it forks a new process for every incoming connection. In the beginning, it parses the `rsyncd.conf` file to determine which modules exist, as well as to properly set any global options configured.

When an incoming connection targets one of the defined modules, the daemon forks off a new child process to handle the connection. Next, this child process reads the `rsyncd.conf` configuration file to properly adjust the options for the requested module. Subsequently, it behaves just like any other rsync server process, adopting either the sender or the receiver role, depending on the case.

3 Design

This chapter is vital to the comprehension of both the purpose and the rationale of the thesis. Here, we unravel the flow of the reasoning that led us to the implementation of the system.

On the first part of the chapter, at first, we briefly present various assumptions and expectations related to the development of our distributed storage system. Then, we thoroughly analyze the fundamental design decisions that we had to make during this process. We gradually unveil how well known concepts and techniques end up playing a major role in our case, sometimes together with background information that helped us sculpt our final opinion about the design. From a practical point of view, this is the chapter in which we mention actual numbers related to the use of the system, whenever deemed necessary, and we do some calculations to argue about our design choices.

On the second part of the chapter, we dive into the architectural and operational details of the system. First, we extract some fundamental principles for the architecture of the system, which hold true for and are applied across all iterations of the development process. Subsequently, building on these principles, we demonstrate how using different algorithms and slight variations of the architectural structure can result in two variants of the initial distributed storage system, both functional, although each pervaded by different qualitative characteristics and ultimately offering slightly different features.

3.1 Assumptions & High-Level Interface

The first step in designing our distributed storage system is to decide the data storage model that will be used to store, retrieve and manage the information.

The system is designed and implemented as a **key-value store**. Data are stored and handled in the form of *key-value pairs*, or **objects**. Each piece of information that needs to be stored, is considered a *Binary Large OBject*, or **blob**, i.e. a collection of binary data that is seen as a single entity. It is also persisted as such, in the form of a byte sequence. Therefore, the system is totally unconcerned about the actual type of the information stored and about what the latter represents for the users, and any further data modeling or schema definition is neither required nor supported. The blob corresponds to the value part of a key-value pair. Every time a new object is created for an incoming blob, the system assigns a unique key to it, in order to form the key-value pair. The system lets the clients know of that key through its response, since the sole method to externally identify and access each blob in the future is via its key.

The objects are stored in one totally **flat namespace**. There are no additional buckets, collections or containers of objects, so there is no way to enforce a hierarchical – or any other – data schema; such needs, when present, should be addressed at a higher level in the architecture of a deployed system that uses ours.

Contrary to various object stores, our current design has not taken into account any needs for storing *metadata* about the objects. Storing metadata is usually considered a useful feature, especially for object-based storage systems, which often support custom object-level metadata capable of capturing subsidiary application-specific information. However, storing such metadata in a resilient manner, and particularly keeping them up to date, is a quite complex procedure, thus outside the scope of this thesis. A careful implementation would require incorporating additional levels to our current proposed architecture, so it fits in well with our thoughts about future expansion of the system's architecture, which are discussed later.

The underlying hardware comprises many inexpensive commodity components, not only for storage, but also for computing and networking. It is expected that many of them will fail, possibly at a most inopportune time. The system must somehow detect,

tolerate and recover from such failures on a routine basis. This is one of the main reasons that Kubernetes is employed, in conjuction with data replication techniques, all of which will be discussed later in this chapter.

The system is designed with a total capacity of about 1 PiB of unique data in mind, i.e. without including the object replicas that will be created if the configurable replication factor is set to a value greater than 1. It stores small blobs, in the order of a few MBs each. Considering blobs of about 4 MiB on average, which is our primarily targeted use case, the system should be able to store around $2^{28} \approx 2.68 \times 10^8$ of them. However, the blobs may be smaller, hence the number of them may be considerably larger. For example, for blobs of an average size of 4 KiB each, our system should be capable of handling around $2^{38} \approx 2.75 \times 10^{11}$ of them, and in some use cases even this number could be exceeded.

Key-value stores usually offer simple and thrifty interfaces to abstract away the lower layers of storage and data management from client applications. So does our system. Since the main purpose of the thesis is to explore containerized storage over Kubernetes, we do not focus on providing a variety of operations on the stored data. Instead, we choose to support merely a small subset of the basic CRUD[1] operations; namely, `Create` and `Read` [2].

The **lack of support for** `Update` operations across the system, hence also for `Delete`, which is usually treated as a special kind of `Update`, needs to be emphasized. `Update` operations during node failures are the main cause of data inconsistencies in distributed storage systems, and one of the main reasons that distributed coordination algorithms and protocols exist in the first place: either simple ones, like 2PC[3] and 3PC[4] [95, 96], or more complex ones, like the Paxos family[119, 120, 121, 122, 123, 124], Raft[125], Zab[126] and Viewstamp Replication[127], they all aim to provide some sort of consensus among a group of participant nodes under various failure models. By disallowing modifications on the stored data, essentially we manage to move past such obstacles, of course at the cost of severely constraining the range of cases in which

[1] Acronym for `Create`, `Read`, `Update` and `Delete`, which are the four basic types of action on persistent storage [94]. Sometimes the acronym may be extended to CRUDL to also cover the `List` operation, which often brings additional complexity on data sets that are too large to hold easily in memory.

[2] `List` has also been studied and designed, but it is implemented only in some versions of the system.

[3] Two-Phase Commit protocol.

[4] Three-Phase Commit protocol.

the use of our system could be an acceptable solution. Furthermore, this choice allows us, not only to focus on the architecture of the system over Kubernetes and Docker, but also to leverage the concepts of **immutability** and **content-addressability** within our design, in a way that will be presented on the following sections of this chapter.

As it should be obvious by now, the workloads consist of `Create` and `Read` operation requests on blobs that either were previously stored, or need to be stored, respectively. Concurrent client requests that refer to the same blobs may be served in parallel by the system, thus the storage engine must be able to handle them properly.

3.2 Communication

Sticking by the design principles of cloud-based applications and systems, clients send their requests and receive the responses over the network; so do the components of the system for their internal communication.

For all sorts of network communication, we use the usual *Internet protocol suite*, also known as TCP/IP stack, as it is implemented in the Linux kernel. The selection has been pretty much trivial to make. IP is the de facto standard protocol on the network layer. On the transport layer, the two most prevalent protocols are TCP and UDP. Transport-layer protocols' most fundamental responsibility is to extend IP's delivery service between two end systems to a delivery service between two operating system processes running on these end systems; this is called *transport-layer multiplexing* and *demultiplexing* [97].

On the one hand, UDP is a connectionless, datagram-oriented protocol that provides the bare minimum services required to be classified as a transport-layer protocol. These are transport-layer multiplexing and demultiplexing, as well as integrity checking by including error-detecting fields in each segment's header. Much like IP, UDP is an unreliable service, as it does not guarantee that data sent by one process will arrive intact – or at all – to the destination process [97]. For that reason, UDP is a popular choice among applications that either require very fine control over what is sent and when, on the application level, or that are able to deal with losing some of the information and value efficiency and fast delivery higher than reliability.

On the other hand, TCP is a connection-based stream-oriented protocol that, first and foremost, provides reliable data transfer over the unreliable IP protocol. To this end it employs a number of sophisticated techniques, including flow control, sequence numbers, acknowledgements, timers, and even a congestion control mechanism to prevent any one TCP connection from swamping the links and routers between the communicating hosts [97]. For that reason, TCP is more complex to implement, and even to configure, but also is the usual choice for applications that require data to be trasferred correctly and in order.

The vastness of configurable features in TCP potentially offers additional benefits to applications and systems that use it. Such applications and systems can properly tune various parameters of their TCP configuration, and implement their own application-level protocols for transferring information over the TCP/IP stack. However, neither designing and implementing an efficient application-level protocol for data transferring nor tuning TCP to such degree is always trivial, and both are certainly outside the scope of this thesis. Therefore, we decided to adhere to one of the most common practices in Microservices Architecture design: to use HTTP on the application layer.

HTTP is the "World Wide Web's application-layer protocol" and it is based on the client-server model. A server receives request messages issued by a client, and it replies with the appropriate generated response messages. HTTP is a *stateless* protocol, in the sense that servers generate responses for incoming requests without storing any state information about the clients. It presumes an underlying and reliable transport protocol, with TCP being the usual choice, as is in our case. The original protocol is `HTTP/1.0` [98], which was later revised to `HTTP/1.1` [99]. Nowadays, it is being, slowly but gradually, superseded by `HTTP/2` [100], which is derived from the `SPDY` [101] protocol, originally developed by Google.

Our system's external API is designed and implemented over HTTP, on the basis of two different paradigms: REST[5] and RPC[6]. Even though both those two paradigms were studied in the context of the thesis, we have to note that the rudimentariness of our system's API does not give the chance for either of them to truly unravel their potential benefits. Nevertheless, the latter of them did stand out for its performance benefits,

[5]Representational State Transfer.
[6]Remote Procedure Call.

thus it turned out to be the sole implementation of our API in the final versions of the system.

3.2.1 REST, JSON & Base64

Representational State Transfer, or *REST* [102], is an architectural style for designing web services and network-based applications. REST-compliant, or *RESTful*, web services expose their state and functionality as a set of resources, and allow the requesting systems to access and manipulate them by the means of a predefined set of stateless operations. When combined with HTTP, these stateless operations are mapped onto HTTP verbs, such as `GET`, `PUT`, `POST`, `DELETE`, etc. All resources are uniquely addressable, most commonly through URIs[7], and the data concerning them are transferred in some known representation, such as HTML, XML, JSON, etc.

Our own primitive RESTful API is designed and implemented over `HTTP/1.1`, using JSON[8][103] as the data-interchange format. JSON does not support binary data natively. To allow the reliable transfer of binary data over media that are designed to deal with textual data, it is necessary to properly encode them first. This gives rise to the issue of choosing an efficient binary-to-text encoding algorithm, and to a subsequent exploration of trade-offs that mainly concern its computational efficiency and the size of the encoded representation that is produced.

We chose Base64[104] as our binary-to-text encoding scheme. It is very popular among RESTful applications that use JSON over HTTP for their APIs and need to transfer binary data, therefore, there is plenty of good implementations to choose from. Each digit of the Base64 encoded output represents exactly 6 bits of data; in other words, every group of three (8-bit) input bytes is represented by four Base64 digits. Therefore, the Base64 representation of a byte sequence is always about 33% larger than the original. Since the encoding, and the decoding, takes place right before sending, and right after receiving each HTTP message, respectively, except from the computational cost of encoding and decoding itself, there is also the non-negligible cost of about 33% additional network bandwidth being used for the data transfer.

More about the design and the implementation of the RESTful API of our system are

[7] Uniform Resource Identifiers.
[8] JavaScript Object Notation.

presented in subsection 4.2.1.

3.2.2 RPC, Protocol Buffers & gRPC

An alternative concept for designing web services and network-based applications is *Remote Procedure Calls*, or *RPC* [105]. It is essentially an extension of the well-known and well-understood abstraction of a procedure call to distributed computing environments [95]. When a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the "callee" environment, i.e. where the procedure is to execute, until the desired procedure is executed and the produced results are passed back to the calling environment, where the execution resumes as if returning from a simple single-machine call [105]. It is, therefore, a form of client-server interaction and a kind of request-response protocol, where the client is the caller that issues the request via its *client stub procedure*, and the server is the callee that produces the response and replies via its *server stub procedure*.

RPC can be used as the central concept of the design of our system's API, as a replacement for REST. Of course, the underlying issues of data transfer are still there. The data still need to be serialized and deserialized for the transfer, and as they can be binary, they also need to be properly encoded and decoded to avoid information loss. Then, they need to be sent over the network using some application-layer protocol. All this needs to be efficient, too. We will now introduce a modern, efficient serialization format as an alternative to JSON, and an RPC framework that makes use of it to achieve great performance without sacrificing simplicity and ease of development.

Protocol buffers, or *protobuf*, is a language-neutral, platform-neutral, extensible way of serializing structured data for use mainly in communication protocols and data storage. It was originally developed by Google for internal use, and was later released under an open-source license [106].

Protocol buffers are flexible, efficient and are accompanied by the `protoc` utility that offers great automatization. With protocol buffers, the developer writes a description of the data structures, or *messages*, in a `.proto` file, using a specialized IDL[9], which

[9]Interface Definition/Description Language.

currently comes in two versions, `proto2` and its successor, `proto3`. Based on that definition, the protocol buffer compiler automatically generates source code, ready-to-use by the application: data structures and functions that implement encoding and parsing of the protocol buffer data. As of May 2018, protobuf supports several programming languages, including Go, Python, Java, C++, C#, Objective-C, Javascript, Ruby, PHP and Dart, though in some cases there are minor incompatibility issues among them.

Protobuf's efficiency stems, not only from its great support for automatization, but also from its encoding scheme. Protocol buffer messages are serialized into a **binary** wire format, which is compact, fast to parse, forward- and backward-compatible, but not self-describing (i.e. the `.proto` definition has to be known for the encoded message to be meaningful, in contrast to JSON or XML that are, to some extent, self-describing). For cases like ours, where blobs need to be transferred over the network, protocol buffers are a significant improvement over Base64 and JSON, both in terms of performance of the serialization and deserialization, and in terms of the network bandwidth that is being used for the transfer.

gRPC is an RPC library and framework, initially developed at Google as a revamp of its internal RPC system, Stubby, and later released under an open-source license.

As many RPC frameworks, it is based around the idea of defining a service and specifying the methods that can be called remotely with their parameters and their return types. The gRPC server implements this interface and handles client calls, whereas a gRPC client has a stub that provides the same methods as the server. The server and the clients can be implemented in a variety of languages – not necessarily the same; currently there is support for Go, Python, Java, C++, C#, Objective-C, Node.js, Ruby, PHP and Dart.

gRPC is essentially a protocol layered over `HTTP/2` [100]. The default choice of data interchange format, which also guarantees high performance and has been tested in many cases across the industry, is protocol buffers. Protocol buffers' IDL is used to define both the data types that are being exchanged between the server and the clients, and the gRPC services themselves along with the methods that they expose. Then, the `protoc` is put into use to generate source code based on the `.proto` definition file. This time, however, it is used with a special gRPC plugin that, except from the regular protocol buffers' code for populating, serializing and retrieving the message

types, additional ready-to-use source code is generated for the gRPC server and client stubs.

More about the design and the implementation of the gRPC API of our system are presented in subsection 4.2.2.

3.3 Immutability

Fundamentally, immutability refers to the simple idea that data remain unmodified after their creation.

As a concept, it is not new at all. It has its roots in the functional programming paradigm, hence its presence in languages like Haskell, Erlang and ML is strong, although it is also present in other programming languages, popular in the industry, like Java, Python and Go, which, for instance, treat strings as immutable objects. Persistent data structures (data structures that always preserve the previous versions of themselves when modified [109], and thus allow queries and modifications to all previous versions of themselves, forming a version tree), which are often harnessed to facilitate parallel and concurrent programming, are usually implemented based on the concept of immutability.

Immutability has also been the basis for VCS[10]. For example, Git's[110] model resembles an on-disk functional data structure with a command line interface to perform operations on it [111]. At its core, it involves *commits*, i.e. independent full snapshots of the entire working directory, forming a *history*, i.e. a version tree, in which individual commits can be referenced using *branches*, i.e. pointers to commits. Actually, Git also supports mutations, i.e. re-writing history, however this practice is controversial and is considered acceptable only under certain circumstances.

The advent of the commoditization of the infrastructure in the datacenters has caused immutability to become relevant in databases and data storage, too. Traditional database management systems were first created in an era that data was small and storage was expensive. The obvious solution used to be to keep the data in records (regardless of the actual data model) and continually mutate it using transactions – this is a model

[10]Version Control Systems.

still flourishing after all these decades. Efficient transaction management has always been a topic of great research interest for both the academia and the industry. In order to squeeze out as much performance as possible, transactions need to execute concurrently. Therefore, the presence of clever techniques for their coordination has always been a necessity, and indeed, quite a few of them have emerged during all these decades. Regardless of whether these techniques are optimistic or pessimistic, i.e. whether they use locks or not, they heavily rely on sets of latches, i.e. low-level locks to provide fine-grained control over internal data structures of the database management system, such as mutexes and semaphores. The use of latches had always been the "less expensive solution".

Nowadays, however, computation keeps getting cheaper thanks to multi-core processors, and so do DRAM[11] chips and SSDs. The coordination of threads within a database system using latches becomes increasingly onerous because of the added latch latency and the consequent loss of instruction opportunities. Meanwhile, the volume of data itself, as well as the number of clients requesting to read or write them, have raised to unimaginable rates – see the explosions of Big Data, IoT[12], etc. So has the need of modern enterprises to deal with it: data need to be stored durably, and retrieved or analyzed reliably and efficiently. Keeping immutable copies of data is now affordable, and could often aid to minimize the latency caused by overwriting, reducing the coordination challenges that crop up in contemporary storage systems [108].

3.3.1 An Alternative Prism

Pat Helland, in [108], argues that many existing ideas, which have been incorporated in the design and implementations of database management systems for years, can be seen through an alternative prism to reveal how immutability really permeates them, too. A *DataSet* is defined as a fixed and immutable set of tables of data, where the actual data model is irrelevant; it can be relational, hierarchical (e.g. JSON), graph-based, or any other. Thanks to immutability, DataSets do not need to be normalized in order to eliminate update anomalies, so the only reason to normalize them would be to reduce the storage necessary for them. Next, he reasons that it is possible for

[11] Dynamic random-access memory.
[12] Internet of Things.

any functional calculation that takes place with a query against data in a traditional relational database, to also take place against a DataSet, since *"when we have snapshots or some form of isolation, database data becomes semantically immutable for the duration of the calculation"*. He further clarifies that there is no semantic obstacle to perform operations like `JOIN` across data stored in relational database management systems and data stored in DataSets, because the "locking" of the data essentially provides a *version* of the database on which such operations may take place. Such locking may be achieved through an isolation mechanism, e.g. based on serializability[128], or snapshot isolation[129], implemented typically using some form of concurrency control, such as 2PL[13] or TO[14], often combined with MVCC[15], all of which are discussed thoroughly in [113, 114].

The notion of versions itself attests to the strong presence of immutability, too. Even in traditional relational database management systems, whenever an ACID[16] transaction is committed, a "new version" of the database is created. These new versions of records and index changes are layered atop previous ones – even deletion, which is often implemented by the means of "tombstone versions". As a matter of a fact, anything mutable can be understood as a set of versions. A key-value store can be built this way; therefore, a relational database management system on top of it as well.

This brings us to LSM-Trees[17], a data structure originally proposed in [130], and popularized thanks to Google's Bigtable paper [131], which also introduced the terms *SSTable* and *memtable*. A SSTable is an immutable on-disk file that contains sorted key-value pairs, whereas a memtable is an in-memory tree-like data structure (e.g. it may be a red-black tree or an AVL tree [132] or even a skip list [133]). Incoming writes are inserted into the memtable, which we can think of as level-0 in the LSM-Tree. Either periodically or when the latter gets bigger than some threshold (usually a few megabytes), it is flushed to disk as a SSTable, which we can think of as level-1 in the LSM-Tree, while incoming writes take place in a new memtable. To serve incoming read requests, the keys are searched first in the memtable (level-0), and if they are not found the search continues to the SSTable files of the next levels, one-

[13] Two-phase locking.
[14] Timestamp Ordering.
[15] Multi-Version Concurrency Control.
[16] ACID stands for Atomicity, Consistency, Isolation, and Durability, which are principles that database designers have historically adhered to for mission-critical transactional systems [113, 114].
[17] Log-Structured Merge-Trees.

by-one. Each level is typically about 10 times as large as its previous. From time to time, the immutable SSTable files that comprise each level are merged and compacted into files of the next level, discarding overwritten and deleted values in the process. As SSTable files are merged, immutable files are read, and brand new immutable files are written; in other words, *"LSM presents a facade of change atop immutable files"* [108]. The technique of LSM-Trees is actually put in use by many storage systems inspired by Google's Bigtable[131], such as Apache HBase[134], Cassandra[19, 20], LevelDB[135], RocksDB[136], WiredTiger[137], and, of course, any other system that uses one of the latter as its storage engine.

3.3.2 The Log

A concept inextricably linked to immutability is the, so called, "append-only" computing, in which information is recorded permanently and forever (practically, for a long time) and derivation of further useful information occurs either on demand or a priori, e.g. periodically. Appending is a sequential write operation, thus is generally much faster than random writes on traditional magnetic spinning-disk hard drives, and also preferable to SSDs [112]. A typical example of this is the *WAL*[18] technique[113, 114], brought into use by the majority of database management systems to ensure durability and recoverability: when a change is made to a database page, a description of the change is written to a separate log file, which lives in stable storage. This file may be called write-ahead log, commit log, transaction log or journal. If the application or system crashes, the log is reviewed during the recovery process. Any changes described in the log that were part of commited transactions but were never written to the actual database itself, are written to the database. Any changes to the actual database, also described in the log, that were part of transactions that were never commited, are backed-out [115]. As noted in [108] (emphasis ours), "from this perspective, the contents of the database hold a caching of the latest record values in the logs. *The truth is the log. The database is a cache of a subset of the log.* That cached subset happens to be the latest value of each record and index value from the log".

The use of an append-only log as the single point of truth is extended to distributed storage systems too, as well as in distributed systems in general. As argued in [116], the

[18] Write-Ahead Logging.

log-centric approach to distributed systems arises from a simple observation that "if two identical, deterministic processes begin in the same state and get the same inputs in the same order, they will produce the same output and end in the same state". The log comes into play when talking about same state, inputs and order. Simply put, the problem of making multiple machines all do the same thing can be reduced to the problem of implementing a distributed consistent log to feed these processed input, hence, the distributed log can be seen as a data structure that models the problem of consensus. These observations are expressions of some of the central notions in *state machine replication* approach[117, 118, 95, 96], which, in fact, is often the basis for consensus protocols, including the popular Paxos family of algorithms[119, 120, 121, 122, 123, 124], Raft[125] and Zab[126].

The idea of distributed log is further used by distributed message brokering systems and distributed streaming platforms. Perhaps the most famous among them is Apache Kafka[139], originally developed at LinkedIn, which maintains a partioned[19] distributed log in a cluster. Each partition is an ordered, immutable sequence of records. It allows producing (by *publishing*) and consuming (by *subscribing*) records to various partitions (called *topics*), by durably persisting all published records using a configurable retention period [140]. A plethora of distributed systems and services have been built atop Apache Kafka thenceforth, all of which transitively leverage this log-centric approach and the underlying notion of immutability. An example is the distributed stream processing framework Apache Samza[141], originally developed at LinkedIn, too. Other projects, which are similar to Apache Kafka and thus underline its value, include the open-source NATS Server[142], a hosted CNCF project, as well as Amazon's commercial service Amazon Kinesis[143].

3.3.3 Immutable Databases

Martin Kleppmann characterizes traditional database management systems as "global, shared, mutable state" [144]. He finds intolerable the fact that, by and large, this situation has not changed since the 1960s, especially since ousting mutable global variables from codebases is already – justifyingly – a ubiquitous practice.

A number of systems actually take a step forward, towards immutability. One of the

[19] Partitioning will be discussed later, in Section 3.6.

most famous among them is Datomic[145], which provides high availability through automatic data replication and failover, strong consistency, a flexible schema, efficient indexing, and supports queries in both the present and past states of the data (and even in hypothetical future states of them) via a declarative logic-based query language. Rich Hickey, one of its designers, argues that past information is immutable by nature, it can never change. Therefore, Datomic's information model is based on accretion of immutable atomic facts, called *datoms*, which may either assert or retract information to modify database's state (this is how queries to future states are performed, too: against temporary in-memory facts, additionally aggregated with those already stored). It is noteworthy that Datomic only aims to impose a certain architecture; the actual storage layer is treated as a black box, and it may be any storage system unless it cannot support key-value semantics or consistent reads [146]. Other storage systems that, one way or another, further exploit immutability's benefits, include Microsoft's Tango[147], CenturyLink's Orchestrate[148], Apache CouchDB[149], as well as RethinkDB[150], especially on its earlier design iterations [151]. Discussing all of them, however, would be out of scope of this thesis.

Of course, as most engineering concepts and techniques do, structuring storage systems on the basis of immutability also has its downsides. A concise critique of this kind is expressed by Baron Schwartz in [152]. He uses Datomic, RethinkDB and CouchDB as references to present his view about two main generic disadvantages of the use of immutability in database systems. First, maintaining access to older facts can be costly in terms of space usage, which keeps growing infinitely, especially since digging up past states of the database is not a frequently used feature in industry applications and their current model. The second is fragmentation: as new immutable facts are accreted, an entity ends up being scattered over a lot of storage space, which may get significantly ineffective in terms of performance, even on SSDs. Solving fragmentation in such environments usually introduces a lot of complexity in both the design and the implementation of the system. He further points out various rough edges and "ugly", yet necessary, design choices imposed by immutability in some of these systems, e.g. the way CouchDB deals with the problem of an underlying disk eventually getting full. Finally, he argues that the use of only partial immutability in traditional relational database management systems, through MVCC to support ACID compliant transactions, as mentioned in the relevant section above, is the result of well-thought

tuning in the design of such architectures, which have matured over all these decades.

All of our design approaches aim to leverage immutability to the fullest, as much as Datomic does – although our data and access models are only a small subset of the respective models of the latter. The component that is responsible for actually storing the data, a microservice actually, does so by keeping them entirely immutable. Data arrive to it in the form of key-value pairs, and are persisted by it as such. Both the keys and the values are immutable sequences of bytes. The keys are of a fixed-size, typically up to a couple of hundred bytes. The values are binary large objects of varying size, up to the order of a few MiB. The system is designed having in mind a total capacity of about 1 PiB of unique data, excluding the configurable number of object replicas that can be created.

3.4 Content-Addressable Storage

Content-addressable storage, also referred to as content-addressed storage or associative storage, is a mechanism for storing information that can be retrieved based on its content instead of its storage location [153]. The following couple of paragraphs are taken from Wikipedia's relevant article[153], which aptly describes content-addressable storage by juxtaposing it with traditional storage mechanisms.

When being contrasted with content-addresssable storage, a typical local or networked storage device is referred to as location-addressed. In a location-addressed storage device, each element of data is stored onto the physical medium, and its location (e.g. path and file names) is recorded to a relevant directory for later use. When a future request is made for a particular item, the request includes only the location of the data. The storage device can then use this information to locate the data on the physical medium, and retrieve it. When new information is written into a location-addressed device, it is simply stored in some available free space, without regard to its content. The information at a given location can usually be altered or completely overwritten without any special action on the part of the storage device [153].

When information is stored into a content-addressable storage system, the system will record a content address, which is an identifier uniquely and permanently linked to the

information content itself. This time, a request to retrieve information from the system must provide the content identifier, from which the system can determine the physical location of the data and retrieve it. Since the identifiers are based on content, any change to a data element will necessarily change its content address. Because of that, in nearly all cases, a content-addressable system will not permit editing information once it has been stored, and whether it can be deleted is often controlled by a policy [153].

By its own definition, it becomes apparent that mingling content-addressability with the concept of immutability is smooth and natural, as a content addressable storage system is usually intended to store data that do not change in time. It is designed to make the searching for some given content very quick, and it can be easily extended to verify that the retrieved data are identical to those originally stored – otherwise the content addresses would differ. In addition, since data is stored into the system by what it contains, there is never a situation where more than one copy of an identical piece of data exists in storage – two such identical pieces have the same content address, by definition, and so point to the same storage location. For data that changes frequently, content-addressable storage is not as efficient as location-based addressing. In these cases, it would be needed to continually recompute the content-based address of the data as they were changed, and the client systems would be forced to continually update information regarding this address. For random access systems, a content-addressable storage system would also need to handle the possibility of two initially identical pieces of data diverging, requiring a copy of one piece to be created on demand.

Nowadays, there are numerous cases where immutable blobs may need to be stored, hence, content-addressability may be deemed particularly useful. For instance, various social networking services host a great number of both text and media files, such as chat histories, images and video. These are perfect candidates for content-addressable storage systems, as they are created once and read multiple times until their deletion – their content is usually not updated at all. In such cases, content-addressability also provides deduplication, effectively relieving storage space consumption, and allowing the appropriate replication and load-balancing mechanisms to handle potential data hot spots, which often crop up in such networks due to social trends, in a planned and efficient manner. The above arguments also apply to the various mail and file

sharing services and platforms that are widely offered in the form of SaaS[20] solutions. Another example where content-addressability can really shine is version controlling: fundamentally, Git[110] is a content-addressable filesystem with a VCS interface. All files' states are stored and given unique addresses based on their content, so as to be possible for them to be referenced later from many file trees, despite being stored only once.

It is obvious that content-addressability makes a great fit in our system's data model and access methods. The lack of support for updates at this level is convenient and qualifies the system as one of the good candidates that were previously discussed. Therefore, our design embraces content-addressability and makes use of it at the core of our system's functionality. Every incoming blob's content is examined and assigned a unique content address, which is actually used as the key in the key-value pairs that are persisted. This key (the content address) is included in the response to the client, so as the client to be able to use it in order to retrieve the stored blob at any point later.

3.5 Hashing

A substantial question that naturally rises and hovers still unanswered over the previous section is: how are those content addresses actually created and assigned to the incoming blobs? The content of each incoming blob is fed into a *hash function*[154] as the input. The produced *digest* (hash function's output byte sequence) is used as the content address: the key in the key-value pairs that are persisted. Hash functions map byte sequences of arbitrary size to ones of fixed size, hence, all digests are of the same size.

Not all hash functions are equivalent and, as expected, the actual choice of a hash function with the needed properties is very important. Content-addressable storage requires that the hash function is *collision resistant*. Otherwise, different pieces of data are subject to collisions, in the sense that they may be assigned to the same key – content address. Collision resistance does not mean that no collisions exist; the pigeonhole principle guarantees that, as long as the input values are bigger in size than the digests,

[20] Software-as-a-Service.

some of them are inevitably hashed to the same output. It simply means that such collisions may take place with very low probability.

For that reason, most content-addressable storage systems expose digests generated by *cryptographic hash functions*[155] from the data they refer to. Those are a special class of hash functions with some additional properties. Some of these properties are that a cryptographic hash function's outputs are deterministic, infeasible to invert, quick to compute, and are governed by the *avalanche effect*[21] [155]. They are also *one-way*[22] and collision resistant [156]. Examples of content-addressable storage systems that use cryptographic hash functions are Git[110], Plan9's Venti[157] and CASPER[158].

An important property of hash functions, with both practical and theoretical value for our case – as it will help us to mathematically model and reason about the problem of selecting a hash function – is *uniformity.* A uniform hash function maps the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability as the rest [154], to formulate a *discrete uniform distribution*[23]. This property is clearly handy when it comes to collision resistance, since a non-uniformly distributed output implies subintervals of high concentration in the total output range, and the probability that any two digests in this subinterval collide is obviously higher.

The size of the produced digests of a hash function is another decisive factor of its collision resistance. It turns out that the collision problem is a generalization of the *birthday problem*[160, 161], which can be stated as follows:

> *Given k random integers drawn from a discrete uniform distribution with range $[0, n-1]$, what is the probability $P(k; n)$ that at least two numbers are the same?*

In our case, the total number of unique assigned content addresses is represented by

[21] Slight modifications of a hash function's input, e.g. the flip of a single bit, cause significant changes to the output. In other words, in principle, similar – but not exactly equal – input values produce totally uncorrelated output values.

[22] Given a hash digest of a N-bit cryptographic hash function, it should require work equivalent to about 2^N hash computations to find any input that hashes to that digest.

[23] In probability theory and statistics, the discrete uniform distribution is a symmetric probability distribution whereby a finite number of values are equally likely to be observed; every one of n values has equal probability $\frac{1}{n}$. Another way of saying "discrete uniform distribution" would be "a known, finite number of outcomes equally likely to happen" [159].

k, while, for a hash function that produces N-bit digests, it would be $n = 2^N$. $P(k; n)$ expresses the probability for a collision between any two content addresses to occur.

It can be proven (e.g. in [160]) that

$$P(k; n) = 1 - \prod_{i=1}^{k-1}(1 - \frac{i}{n}) \tag{3.1}$$

when $k \leq n$, with

$$P(k; n) \approx 1 - e^{-\frac{k(k-1)}{2n}} \tag{3.2}$$

being a good approximation.

Alternatively, for simplicity, we can think of probability $P(k; n)$ to be bounded by the number of all pairs of random integers multiplied by the probability that a given pair will collide [157], i.e.

$$P(k; n) \leq \frac{k(k-1)}{2} \times \frac{1}{n} \tag{3.3}$$

We start by examining the range of values for k in our case. Considering a total capacity of 1 PiB of unique data objects, and each one of them being about 4 MiB on average, our system should be able to accomodate at least $2^{28} \approx 2.7 \times 10^8$ objects. Of course, as some of the blobs may be smaller, their number can get quite larger. For example, for objects in the order of 4 KiB each on average, the system should be able to accomodate about $2^{38} \approx 2.75 \times 10^{11}$ of them.

Next, we examine the values for n. As it was previously mentioned, $n = 2^N$ for any hash function that produces output of N bits. To begin with, we intuitively rule out all 32-bit hash functions, since $n = 2^{32}$ is plainly inadequate, even for the lowest anticipated values of k ($\approx 2^{28}$ objects). Other than those, there are plenty of uniform hash functions of 64-bit digests – or more – to choose from. It should be noted at this point, however, that there is no "free lunch" considering any of these choices, really. Since the N-bit digests are used as keys in the key-value pairs, one way or another they need to be stored in the system as well. In an extreme case, where $k \approx 2^{38}$ and a collision-resistant 512-bit uniform hash function is selected, the storage space required only for the keys is as much as 16 TiB. This is about 1.56% extra to total system's capacity, which is tremendously high, especially considering that the keys themselves carry no semantic or other useful to the users information, and that their sole purpose is to

facilitate the implementation of content-addressability in the system. Therefore, intuitively, we may also rule out all 512-bit hash functions, since, on the one hand $n = 2^{512}$ looks unnecessarily excessive, even for the largest anticipated values of k, and on the other hand the trade-off for very low probability of collisions, i.e. the cost of storing the digests along with the blobs, can be considered imbalanced for a plethora of use cases.

Based on these observations, using one of (3.1), (3.2), (3.3), it is now possible to calculate the probability of collisions for different combinations of values of n and k. The calculations include values of N in the range $[64, 384]$ that are widely used by many hash functions, and two significant values of k. They are shown in Table 3.1.

N	**n**	**P(k; n)**	
		k = $2^{28} \approx 2.68 \times 10^8$	**k = $2^{38} \approx 2.75 \times 10^{11}$**
64	$2^{64} \approx 1.8 \times 10^{19}$	2×10^{-3}	1
128	$2^{128} \approx 3.4 \times 10^{38}$	1.06×10^{-22}	1.11×10^{-16}
160	$2^{160} \approx 1.5 \times 10^{48}$	2.47×10^{-32}	2.58×10^{-26}
224	$2^{224} \approx 2.7 \times 10^{67}$	1.34×10^{-51}	1.4×10^{-45}
256	$2^{256} \approx 1.2 \times 10^{77}$	3.11×10^{-61}	3.26×10^{-55}
384	$2^{384} \approx 3.9 \times 10^{115}$	9.14×10^{-100}	9.59×10^{-94}

Table 3.1: *The probability of collisions between any two hash digests, $P(k; n)$, for widely used values of n and a couple of values of k.*

Conversely, from (3.2) it is possible to derive an approximation formula for k, the number of objects stored in the system, given a N-bit uniform hash function, and a desirable value for the probability of collisions between any two digests in range $[0, n - 1]$, when $k \leq n$:

$$k(P; n) \approx \sqrt{2n \ln\left(\frac{1}{1 - P}\right)} \tag{3.4}$$

where $n = 2^N$.

Next, using (3.4) and based on the above values, it is possible to calculate the maximum values of k, for the same values of N and n, given that $P = 10^{-18}$ is the desired probability of such collision to occur. The results are presented in Table 3.2.

The probability $P = 10^{-18}$ is chosen for comparison, based on the fact that 10^{-18} to 10^{-15} is the uncorrectable bit error rate of a typical hard disk [162, 163].

By skimming through both tables, it is obvious that 64-bit hash functions cannot be

N	n	k
64	$2^{64} \approx 1.8 \times 10^{19}$	12
128	$2^{128} \approx 3.4 \times 10^{38}$	$2.6 \times 10^{10} \approx 1.51 \times 2^{34}$
160	$2^{160} \approx 1.5 \times 10^{48}$	$1.7 \times 10^{15} \approx 1.5 \times 2^{50}$
224	$2^{224} \approx 2.7 \times 10^{67}$	$7.34 \times 10^{24} \approx 1.51 \times 2^{82}$
256	$2^{256} \approx 1.2 \times 10^{77}$	$4.81 \times 10^{29} \approx 1.51 \times 2^{98}$
384	$2^{384} \approx 3.9 \times 10^{115}$	$8.87 \times 10^{48} \approx 1.51 \times 2^{162}$

Table 3.2: *Safe-to-use values of $k(P; n)$, i.e. the maximum number of objects stored in the system, for widely used values of n, given that the desired probability of collision between any two hash digests is $P = 10^{-18}$.*

trusted to avoid collisions for the required maximum number of stored objects in our content-addressable storage system. Even if we reduce the desired value of the probability that any two hash digests to collide to as low as $P = 10^{-10}$, it would not be safe to store more than about $60000 < 2^{16}$ objects, which is very low in general.

On the other hand, 384-bit, but even 224-bit and 256-bit, hash functions seem to exceed the requirements for our purpose. Although diminishing the probability of collisions as much as possible is definitely desirable, we ought to mind the imposed storage space tradeoff, which becomes vexatious proportionally to the chosen size of hash digests. Hash functions with 128-bit and 160-bit output may look right on the edge: although they may both be safe enough to be used for $k \approx 2.68 \times 10^8$ objects stored, for values of k in the order of $\sim 2^{48} \approx 2.8 \times 10^{14}$ selecting one of them is worrisome.

Before selecting a hash function, we might have to consider our choice from a security perspective. A one-way collision resistant hash function, like one of the cryptographic ones, can be employed to provide some security-related assurances to both the system and its clients. On the system's side, it can prevent potential malicious clients from intentionally creating objects that violate the assumption that each object has a unique key. From a client's point of view, it allows for stronger integrity checks, preventing a malicious server from fulfilling a read request with fraudulent data.

From all the above, we infer that choosing an appropriate hash function depends heavily on the requirements imposed by each specific use case of the system, which are possibly deduced by a higher-level business logic or architecture each time. Our storage system is designed and implemented to function properly regardless of the actual hash function being chosen. Even though collision resistance, hence content-addressability,

does rely on uniformity and the size of the produced hash digests, every other aspect of the design, concerning various components of the system, is impertinent to the selection of a hash function. This design allows for effortless upgradability of the system, providing flexibility across different use cases. More on hashing and the actual (current) implementation are discussed in Section 4.3.

3.6 Sharding

So far we have discussed about immutability, content-addressable storage and hashing, and how they fit in with the design of our storage system. None of these, however, have anything to do with distributed computing-specific concepts. In this section, we present another major concept in the core of our system's design, which is widely perceived as an essential pillar of horizontal scalability and elasticity in modern distributed storage systems.

Database partitioning is the act of breaking up a logical database or its constituent elements into distinct independent parts for reasons of performance, availability, load balancing or simply manageability [177]. Traditionally, partitioning can be either *vertical* or *horizontal*, but we will not deal with the former at all. In **horizontal partitioning** the data are partitioned into multiple different units, typically based on a key – usually each record's or object's key. There are numerous advantages to this approach, mostly from a performance point of view: index sizes are reduced, and queries can be directed to the appropriate partition, thus search performance is improved.

Sharding is a special case of horizontal partitioning where the partitions, now also called **shards**, can be distributed among multiple physical nodes in different locations. These nodes should be able to function properly even when they are isolated, which requires more care than simple horizontal partitioning, as the hoped-for gains in efficiency would be lost, if querying the system required multiple instances to be queried [178]. This is also the reason that sharding is often related to a *shared-nothing architecture*[24] – once sharded, each shard should be able to function separately from the rest. This adds **horizontal scalability** to the list of benefits from sharding, since dif-

[24] A shared-nothing architecture is a distributed-computing architecture in which each node is independent and self-sufficient – nodes do not share memory or disk storage with each other, and there is no single point of contention across the system [179].

ferent machines can each be dedicated to store and serve queries to a specific shard – or more of them – and the number of the machines can be adjusted to the requisite scale. Furthermore, the presumed isolation in the architecture usually helps reasoning about the replication technique, when required. The partitions may be either disjoint, i.e. every piece of data in the storage system appears only in a single shard and its assigned server node acts as the single source for it, or some complex replication schema may be applied, in which multiple replicas of a record or an object can be present in multiple shards.

Our distributed storage system is designed to run on commodity hard disk drives and SSDs. Nowadays, the typical capacity of such a disk is about 2 TiB each, which may yield a total storage capacity of about 16 TiB per datacenter node. It is obvious that, for our system to be capable to accomodate 1 PiB of data, the data must be sharded across multiple nodes. For example, even if no additional replicas are required for each object, in order to persist 1 PiB of data on nodes of 16 TiB storage capacity, at least 64 of them are needed. Therefore, the minimum number of shards that should be created, leaving replication aside for the time being, would be 64. Taking into account a replication factor of k, our system would need to be deployed on $64 \cdot k$ nodes in the datacenter, and the actual placement of the shards across these nodes should be treated with special care, since colocated replicas of an object would not effectively contribute to resilience against disk or node failures.

A critical issue around sharding is how to properly **balance the load** of the system across the nodes. In other words, in which node should each object be stored so that, while all nodes are independent from one another, all objects are distributed across the nodes in a balanced way? And how is "balanced" defined in each case? In practice, the storage capacity of each node in the datacenter may vary, especially in on-premises environments. Ideally, the objects should be distributed proportionally to each node's storage capacity; therefore, the size of the shard that is assigned to each node should correspond to that node's capacity. Practically, this is perfectly feasible, but somewhat complex. In the context of this thesis, we have developed two systems that each employs a different sharding technique. To overcome this matter, in both systems we assume that the storage capacity of every node is equal [25].

[25] However, one of the sharding techniques is designed and implemented in such a way that confines

Another major issue related sharding techniques, is their behavior when nodes must join or be removed from the cluster. **Resharding** is the action of redividing an already sharded logical database into new shards. The need for resharding may emerge for a number of reasons; for instance, what happens when the number of shards that the storage system has been splitted into is equal to the number of nodes in the cluster where the system is deployed, but the total storage capacity needs to be further increased by deploying the system to a number of new cluster nodes? There is obviously a need to further split the data into more shards; hence, the exact rules of the sharding process for this system are in some way changed from that time on. How does this change reflect in the data that are already present in the current shards? As the shards are allocated to physical nodes, their data are bound to the underlying nodes; moving them around can be costly in terms of network bandwidth and performance in general, and, of course, catastrophic if implemented incautiously. What portion of the existing data gets to be moved to the new shards? During this data movement, do data flow only towards the newly added nodes, or between the existing ones, too? Are the rules of the replication schema affected by resharding, and if yes, then in what way? These are only some of the core, high-level questions that a sharding technique needs to answer regarding resharding – there is no single correct answer to any of them, and different policies may fit well for different use cases.

Among various approaches for carrying out sharding, we chose ones that use hashing as their basis. Hashing is already being used to provide content-addressability, so it is already present in the core of our system; utilizing it for another purpose does not create any additional impediments, performance or otherwise. By the means of a uniform N-bit hash function, the objects' keys – hash digests – are uniformly distributed values in range $[0, 2^N - 1]$, our *keyspace*. Sharding can be examined as the problem of merely finding an appropriate segmentation of the keyspace. The segmentation itself, as well as the process of assigning each of these segments to a server node, varies between the different sharding techniques that were employed by our two systems. So do their capability and behavior regarding resharding. The specifics of these techniques, along with their advantages and disadvantages, will be discussed in the following sections of this chapter, where each system's design is presented separately.

this issue within one of its constituent libraries. Expanding this library, in a way that will be discussed later in section 5.2, in conjunction with the proper configuration of Kubernetes, would allow finer-grained, sophisticated sharding, proportionally to each node's storage capacity.

3.7 Architecture – A 10,000 feet view

Before diving into each one of the two systems to explain their design and their differences, it is deemed constructive to point out their similarities first. So far in this chapter, we have been discussing techniques, methods and concepts that are present in many modern distributed storage systems, and also play a major part in our own systems' design. At this point, it is time to assemble all those pieces together to form the common architecture shared by both systems, from a high-level perspective.

The influence of Microservices Architecture's principles and patterns upon our design is conspicuous. The proposed architecture is **multi-tiered**: there is a frontend and a backend tier. On each tier, a variable number of instances may be deployed. The instances of the frontend tier are the **Proxies** and these of the backend tier are the **Agents**; they may henceforth be referred to as such, interchangeably with the frontend and backend terminology.

Figure 3.1 illustrates a high-level overview of the proposed architecture that is independant of the sharding and replication techniques employed by each system, thus it is common to both of them. Next, we begin by abstractly describing the responsibilities of the basic components.

3.7.1 Proxies — the Frontend

The frontend tier has primarily two responsibilities. First, it is the one communicating with the clients: it is the first component to receive their requests and the one that replies to them with the final response. The system's external API, which was abstractly adumbrated in section 3.1, and may be based either on REST or on RPC, is implemented and exposed by each Proxy. The second responsibility of the Proxies is that they act – unsurprisingly – as reverse proxy servers for the incoming client requests, forwarding them to the appropriate Agents and aggregating the upstream results before replying to the clients. Deciding on which Agents should each client request be forwarded to, is dictated by the sharding technique that is being put in use throughout the whole system; thus, Proxies' role in the process of sharding is paramount. Moreover, as the Agents that a client request should be forwarded to can be more than one, it is implied that part of the replication algorithm is executed on the frontend tier, too.

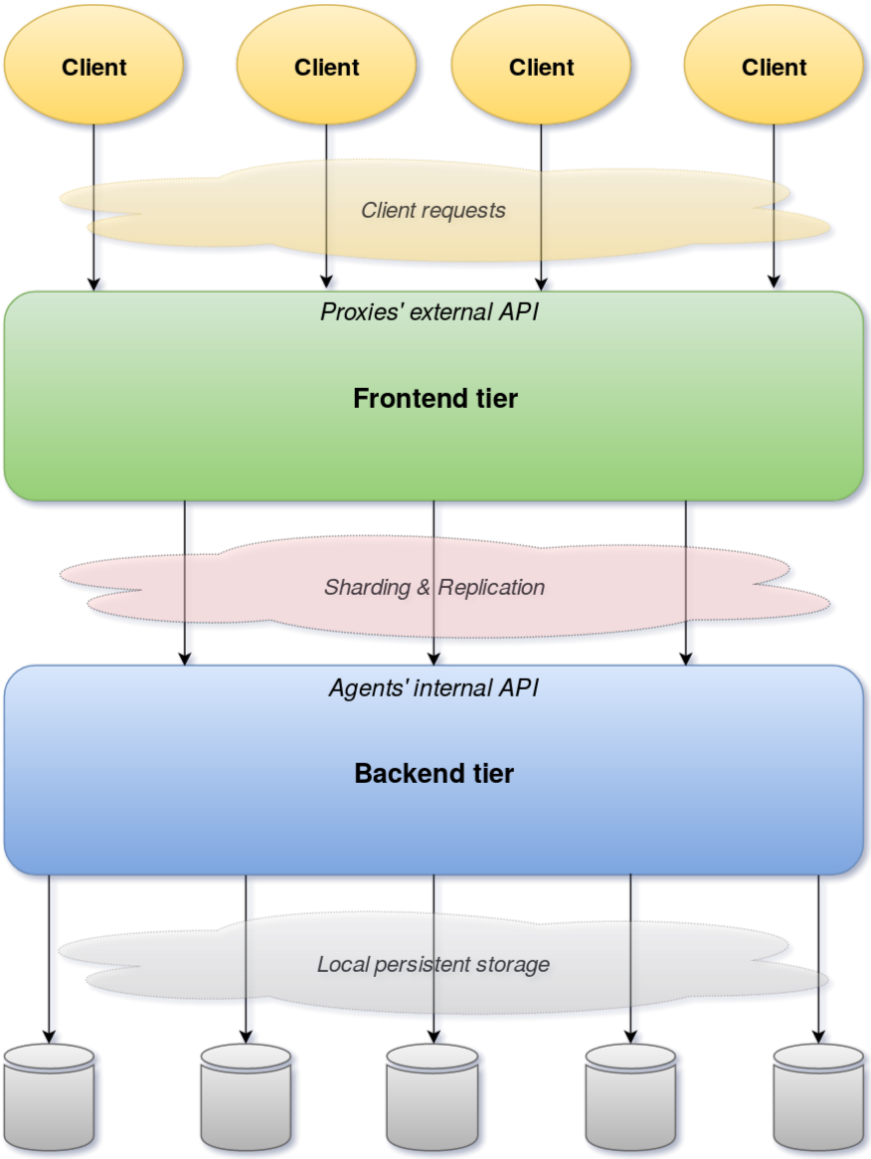


Figure 3.1: *High-level illustration of the proposed architecture, common to both systems.*

This is expected anyway, since the replication scheme is usually deeply intertwined with the sharding method.

An important trait of the frontend tier is that Proxies are **stateless**. Since it is not them storing the data, there is no state information that needs to be persisted durably at all. A Proxy's "state" is limited to the pending client requests that need to be handled at any point in time, which is, of course, temporary. This sort of state information is temporarily stored in memory until the handling of the request is considered finished; there is no need to access the underlying storage hardware at all. Consequently, disk failures do not cause any trouble for the frontend tier itself, under any circumstances.

The statelessness in the design of the Proxies provides significant benefits regarding their scalability and elasticity; the frontend tier can be scaled both up or down, and out or in. Proxies are vertically scalable – to a certain extent – since all they really require is computational power and memory. Themselves being multi-threaded, so that it is possible for them to handle multiple concurrent client requests in parallel [26], Proxies are **vertically scalable**: just stacking as many CPU cores and as much DRAM capacity as possible to the underlying nodes translates directly into enhanced capability of handling a greater number of client requests in parallel.

The frontend tier is **horizontally scalable**, too: merely deploying the required number of Proxy instances on the cluster nodes suffices. The statelessness in Proxies' design ensures that they have no identity nor previous state to retrieve or synchronize. Therefore, the process of scaling them in or out is fairly trivial: one may simply spawn or kill as many of them as required. As a matter of a fact, this can take place at any time during the operation of the system, e.g. depending on its load, thus ensuring absolute **elasticity** for the frontend tier. It also ensures that Proxies are **resilient to node failures**, in the sense that, although some of them may disappear at any point in time and, therefore, their pending client requests may need to be restarted as well, such failures do not cause any disruption to other components of the system, nor is it disconcerting to simply deploy a new instance on some other cluster node immediately. On that account, even though Proxies are a crucial part of the system and they are always on the datapath, they are normally not expected to be the bottleneck in system's overall throughput, or performance in general, by design.

3.7.2 Agents — the Backend

The backend tier is primarily responsible for persisting the blobs. It is this tier where durability must be ensured, and where internal indexing and concurrent object operations must be handled; hence, it is where the storage engine lives. Agents implement a simple, separate API, which is not exposed externally to the clients, but it is accessible by the Proxies. It, too, may be based either on REST or on RPC. When Proxies receive the client requests and decide on which of the Agents they should be forwarded to, this

[26] In fact, the implementation uses lightweight user-space threads wherever possible, e.g. goroutines in Go, and `gevent` module's greenlets in Python.

forwarding takes place by issuing operations that are defined in that internal API and implemented by every Agent.

The fact that Agents are responsible for storing the objects is what makes them inherently **stateful**; and this is, as usual, a burden. First of all, Agents' state needs to be persisted durably; therefore, they rely heavily on their underlying storage hardware, and except from network failures, and any other type of failure that may render a node nonfunctional, Agents are also susceptible to disk failures. Given the fact that they are stateful, though, the process of failing them over is not as simple as it is in the case of the Proxies, because Agents must somehow "carry around" the data they are responsible for, which may well be in the order of TBs. A "disappearing" Agent, especially when no replication algorithm is in effect, essentially translates into unavailability of a portion of the data, which is an unacceptable behavior for any storage system.

Applying a **replication** scheme is indeed a significant step towards improving the situation at hand. Yet, even when the objects are persisted via multiple replicas across many Agents, in the case of an unrecoverable node failure where an Agent would need to be restarted on some other cluster node, it must somehow redeem the data it is responsible for, as quickly as possible, in order to be fully functional again. The data are redeemed with the assistance of the other Agents that persist replicas of the same data. During a period that an Agent's stored data are not in "a ready state", not only a portion of the data of a shard has reduced availability, but there is also increased danger of complete data loss: assuming a replication factor of k, if all k Agents that persist replicas of the objects in a shard happen to fail, none of them will be able to restore them upon re-joining the cluster. In general, under our failure model, **replication factor of k guarantees $k-1$ fault tolerance** in a well designed system, i.e. no more than $k-1$ Agents owning replicas of a shard may be in a "failed state" simultaneously without incurring data loss. At any point in time, the more Agents are in a non-ready state regarding their data stored, the more impaired is the availability of the data in the affected shards, and the fewer additional failures can be tolerated during this period.

There are many ways to implement the replication mechanism of a distributed storage system. First of all, the replication model can be either *active* or *passive*. In the passive replication model, the frontend interacts with a single *primary replica manager*, a *master*. The master is responsible for responding to requests on behalf of the backend,

and also inform a number of *secondary replica managers*, or *slaves*, about changes of the state stored. In the event that the master fails, one of the *slaves* becomes the new master. Sticking by the principles of state machine replication, our systems' data replication mechanism is based on the **active replication model**. According to this model there is no single primary replica manager for every object stored; all replica managers, i.e. the Agents in our systems, are equivalent. The frontend forwards the requests to all these Agents, and each Agent executes them independently. The results of the executions are accumulated by the Proxies, which then decide on the final response of the system to the client.

Consequently, one of the most important – and non-trivial to get right – responsibilities of an Agent is its contribution to the overall resilience of the system to failures, despite its inherent statefulness. An Agent must be aware of most aspects of the sharding and replication techniques that are being used throughout the system. It needs to know at least the exact number of shards that exist across the system at any time, as well as all the other Agents that are responsible to store replicas of the data in the shard – or shards – that itself stores, too. It must be able to reach them upon failing over, so as to retrieve the replicas of the data that it should own – using some sort of bulk loading – and also be available to being reached by such Agents for the same reason.

When a cluster of Agents is deployed for the first time, we say that it is in the **Bootstrapping phase**. An Agent that begins its operation at a time that the Agent cluster is not in the Bootstrapping phase, can only be restarting from a failure. In this case, as it has already been explained, it seeks to recover the data it should persist, as fast as possible. We call this process **initial data synchronization**, or *initial synchronization*, or sometimes `initsync` for brevity. We say that an Agent during its initial data synchronization process is in the **Syncing phase**, or **Sync phase**, or **Init Sync phase**. When the initial data synchronization is finished, the Agent usually proceeds to the **Ready phase**, which means that it is completely up to date regarding its stored data and ready to serve any forwarded request incoming from the Proxies. Actually, for one of the two systems there are more *phases* that the Agent cluster can reach, but all these details, as well as the conditions for transitioning from each one of them to another will be discussed thoroughly in time.

During the initial data synchronization, the transfer of the data must meet some cri-

teria. First, it must take place in a reliable way; hence, as discussed in section 3.2, TCP is probably the first choice of a transport-layer protocol that comes to mind. Furthermore, we have to consider that a restarting Agent may, in fact, still own some part of its data – or even all of them – if, for any reason, it is rescheduled onto the same cluster node after the failure. We need a "clever" way to determine the portion of data that is up to date and the portion of them that must be retrieved from other Agents.

Last, we have to decide on the direction of the data flow during the transfer. The *pull-based* model would make the transfer of the data a responsibility of the restarting Agent: it would be the Agent that should ask other Agents for the portions of the data he does not have upon restarting from a failure, and *pull*, or *download*, them. The *push-based model* would make the transfer of the data a responsibility of the healthy and up-to-date Agents: it would be them that should keep track of freshly-started Agents and *push*, or *upload*, to them any data that they may possibly be missing. In our case, the **pull-based** model seems to fit more naturally into the architecture, and is therefore selected: every Agent is always expected to listen for initial synchronization requests from failed-over Agents that ask to pull any data they should be storing, and to serve them to the best of its ability. Designing and implementing a specialized application-layer protocol to efficiently address all those issues – and those only, is often the best solution in situations like this. However, designing and implementing a custom protocol so that it both is correct and provides higher performance than the great variety of already existing, battle-tested protocols out there, is not a trivial procedure, and certainly not an immediate concern of this thesis. Thus, the data transfer is entrusted to one of these protocols. Specifically, **rsync**, with the proper tuning, looks like a perfect fit for our case, but the details are postponed until later in this chapter, as well as section 4.5.

3.7.3 Kubernetes

At this point, we are in a position to abstractly map some of the components and the functionality of our system onto Kubernetes API objects.

The frontend tier, comprising a variable number of Proxies, requires merely a small set of Kubernetes' features. In particular, it requires Kubernetes' most basic functionality: that it tracks each Proxy's running status to make sure it is up and serving client re-

quests, and also that it intervenes on failures by spawning new Proxy instances on eligible cluster nodes. The deployment unit of Kubernetes is the Pod; hence, it is straightforward that each Proxy instance should probably be deployed in a single-container Pod, i.e. a Kubernetes Pod that encompasses only one Docker container, which runs a single process of the Proxy binary.

There are more than one options for the higher-level Kubernetes API object that keeps track of the Proxy Pods. Considering that Proxies are entirely stateless and that they have to keep serving client requests forever – until they are stopped or they fail, it turns out we can use either a `Deployment` or a `DaemonSet`. The exact choice between those two, depends on the composition of the cluster and the specific needs of the deployment. Their core difference is that a `DaemonSet` ensures that all Kubernetes cluster `Nodes` run a copy of the Pod at any time, whereas a `Deployment` ensures that a specific number of Pods is running, on any cluster `Nodes`, at any time. Both of these two options respect Kubernetes' node affinity configuration [180], therefore, specific nodes can be included in – or excluded from – the set of candidate nodes during the Pod scheduling process. It is obvious, though, that, most of the times, `Deployment` is the most flexible option, as it makes it as easy as possible to scale in or out the frontend tier, by simply setting the desired number of Pod replicas that should be running.

On the backend tier, which consists of a variable number of Agents, the requirements are stricter. First of all, Kubernetes' core features are still required: Kubernetes should monitor the running status of each Agent and make sure to intervene when something goes wrong, by restarting the failed ones on alternative eligible cluster nodes. This time, however, contrary to the case of Proxies, since each Agent is tied to the data it stores as well as to the responsibility of always keeping them intact and up to date, every Agent can benefit from having an identity that is unique among its peers-Agents. Moreover, as the Agents may often need to communicate with one another to synchronize their stored data – in fact, each with specific others – they can also benefit from having their unique identity also tied to a stable network address, as a means to contact any Agent at any time, as long as it is not in a failed state, of course. These are features that can be provided via a `StatefulSet`, a Kubernetes API object whose original purpose matches exactly our case: it is supposed to to be used for deploying and managing stateful applications and systems.

Similarly to `Deployments` and `DaemonSets`, `StatefulSets` are higher-level API objects that keep track of sets of underlying Pods. As in the case of Proxies, each Agent is assigned to a single Pod. Only this time, the Pod encompasses two Docker containers, instead of a single one. The first container is where the actual Agent process runs, whereas the second includes an always-running `rsync` server running in daemon mode, awaiting to serve other Agents' requests for data synchronization. The specifics of the use of `StatefulSets` differ between the two systems, so the discussion about them is postponed until each system is presented.

It is obvious by now that communication plays a large part in the system's functionality. This includes the communication among the deployed components of the system across both tiers, as well as the communication between the system and its clients. However, it is already known that the environment of most modern datacenters, which are mostly comprised of commodity hardware, makes the deployed containers prone to failures, especially as the needs of the system are increased, along with the size and the complexity of the cluster. As various instances of the frontend and the backend fail and are restarted on different nodes, they need to be able to reach – and also themselves be reached by – other components of the system again, even though the network locations (IP addresses and ports) may change all the time. The same problem occurs in cases that a scale-out, or a scale-in, causes additional such instances to be created, or destroyed, respectively. This is a rough description of a common problem in many distributed systems, known as **service discovery**.

Kubernetes provides an automated solution for service discovery through its `Services` and `Endpoints` API objects [84]. They are backed by an underlying mechanism, which may vary among different Kubernetes clusters, that carefully assigns cluster-internal, virtual IP addresses to the deployed Pods; in our case it is always an overlay network via `flannel`[185] running in VXLAN mode. `Services` themselves are a mechanism whereby a set of Pods, determined using Kubernetes `Labels` and `Selectors`, is targeted, and access to them is allowed conveniently, without the need to know their IP address, which may often be changing dynamically. Similarly in our case, the Pods controlled by the higher-level Kubernetes API objects that are being used, i.e. the `Deployment` or `DaemonSet` and the `StatefulSets`, expose various APIs, which are accessible thanks to the use of `Services`.

Proxies must be able to communicate with the clients, which may well live outside the cluster. In other words, Proxies' API must be exposed "externally", i.e. outside of the Kubernetes cluster. To this end, two types of Kubernetes `Services` can be used: `NodePort` and `LoadBalancer`. The choice between these two depends on the cluster configuration each time, e.g. on the features of the cluster's underlying cloud provider. From now on, we assume that the frontend tier's API is exposed via a `Service` of the `NodePort` type, which is the most generic option, allowing us to argue about the system ignoring any special – often facilitating – factor.

Agents must be able to communicate with the Proxies via their own API in order to handle the forwarded client requests, and also with their peer Agents for both issuing and serving initial data synchronization requests. Consequently, there is no need for them to communicate with anything outside the cluster. Considering that the backend tier makes use of `StatefulSets`, which always require a `Headless Service` as they need neither load balancing nor a single service IP address, for the communication of each `StatefulSet` that is deployed for the Agents with any other component of the system, one `Headless Service` should suffice.

Last but not least, Kubernetes is responsible for providing local persistent storage to the Agent Pods, either via `Volumes` [186] or via `PersistentVolumes` and `PersistentVolumeClaims` [187]. Thanks to its rapid development cycles, at the time of writing this thesis, Kubernetes supports the provision of local peristent storage to Pods via statically created `PersistentVolumes`, although dynamic provisioning is not supported yet. This option offers many advantages, but also comes with some constraints. Analyzing them is out of the scope of the thesis though, because at the time of designing and implementing our systems, the options for providing local persistent storage to the Agent Pods via Kubernetes were quite limited. For that reason, we ended up using one of the simplest Kubernetes `Volume` types, the `emptyDir`.

An `emptyDir` is first created when a Pod is assigned to a Kubernetes cluster node, and exists as long as that Pod is running on that node. As the name implies, it is initially empty. All containers in the Pod can read and write the same files in the `emptyDir` volume, though that volume can be mounted at different paths in each container. When a Pod is removed from a node, for any reason, the data in the `emptyDir` is deleted forever, and this sometimes complicates our situation, and makes this solution rough and

temporary. Provisioning local persistent storage for the Agent Pods using the relevant new feature of Kubernetes is, in fact, one of the prime concerns of the future work that has to be done regarding our systems.

As it has been mentioned repeatedly, the design process took place in multiple iterations, resulting in two distinct systems. In the previous sections, by referring to "the system" we actually referred to both of them. Indeed, all concepts that have been discussed so far, as well as the high-level architecture described in this section, apply to both of them. However, they support different features, each offering different advantages and disadvantages. All these will be presented in the following sections, where `MICAS` and `RICAS`, the two systems are introduced and discussed separately. From now on, each system will be referred to either as "the system" when the context is clear, or explicitly by its name when the context is ambiguous.

3.8 MICAS

`MICAS`, which stands for Modulo-based Immutable Content-Addressable Storage, is the result of our initial design approach, and the first of the two systems that are presented in this thesis.

3.8.1 Sharding using mod-N Hashing

The sharding method used by `MICAS` is one of the easiest ones to perceive and explain. From now on, it will be referred to as **mod-N hashing**.

The whole concept is pretty simple. We begin by selecting h, a N_h-bit hash function, taking into account the analysis in section 3.5. Upon each object's creation, the content of the incoming blob is given as an input to the hash function. The output, a N_h bit long hash digest, is the content address of the object that will be stored for that blob, and thus also its key, as explained in section 3.4.

Next, we have to decide on a **static number of shards**, N_s, into which the keyspace should be splitted. The case of using a dynamic number of shards and the behavior

of *resharding* will be examined a bit later. Each shard is assigned a unique ID, from the integers in range $[0, N_s - 1]$. The shard, S_i, on which the blob b_i, is assigned to, is derived from the simple calculation:

$$S_i = h(b_i) \bmod N_s \tag{3.5}$$

where $h(b_i)$ represents the application of the hash function h on blob b_i's content, i.e. the resulting hash digest – object's key – for the blob b_i. The use of the modulo operation is where the system's name is derived from, by the way.

So, what is the role of modulo in this calculation, anyway? In general, computing modulo between various integers and a fixed divisor can be regarded as arranging those integers into groups, as many as the value of the divisor, where these groups are labeled by all possible results of the modulo operation: the integers in the range $[0, divisor-1]$. Usually, this is even more intuitive – as well as computationally efficient – in cases that the divisor is a power of two, e.g. 2^e: the modulo operation can be computed merely by applying a mask on the e least significant bits of the dividend. Back to our situation, as long as the selected hash function h is uniform, i.e. the probability that it generates any hash digest is roughly the same for all of them, the modulo operation ensures that the integers in h's range, i.e. the keyspace $[0, 2^{N_h} - 1]$, can be arranged almost uniformly in such groups. In fact, the grouping is perfectly uniform if and only if the divisor is a power of two, e.g. $d = 2^e$, and each group consists of exactly $2^{N_h - e}$ integers in the keyspace. Otherwise, if $2 \nmid d$, a number of these groups, $g = 2^{N_h} \bmod d$, consist of one fewer integer than the rest of $d - g$ groups, a difference that is anyway negligible in the order of $\lfloor \frac{2^{N_h}}{d} \rfloor$ for any reasonably large value of N_h.

A reasonable analogy is to consider the shards as *bins* and the hash digests generated from the incoming blobs as *balls*. There is a total of N_s bins, and the balls are "randomly" allocated to them so that each ball has probability $\frac{1}{N_s}$ of falling into a bin. As a matter of a fact, the classic "balls and bins" problem in probability theory is a common way to model load balancing problems, such as those related to sharding. However, such a probabilistic and statistical analysis of load balancing is beyond the scope of this thesis, and the reader is referred to [181, 182] as starting points for more on the topic. In our case, the balls, i.e. all generated hash digests from the incoming blobs for the whole duration of the deployment of the system, let n be their cardinality, can

be considered as a random sample drawn from the population of all integers in the keyspace $[0, 2^{N_h} - 1]$. We content ourselves with the intuition that each one of the N_s bins should be filled with about $\frac{n}{N_s}$ balls. Intuitively again, we understand that, for large values of n, e.g. in the order of 2^{20} or higher, when load balancing really matters in terms of storage space, the variation of the number of balls per bin around $\frac{n}{N_s}$ should tend to be normally distributed.

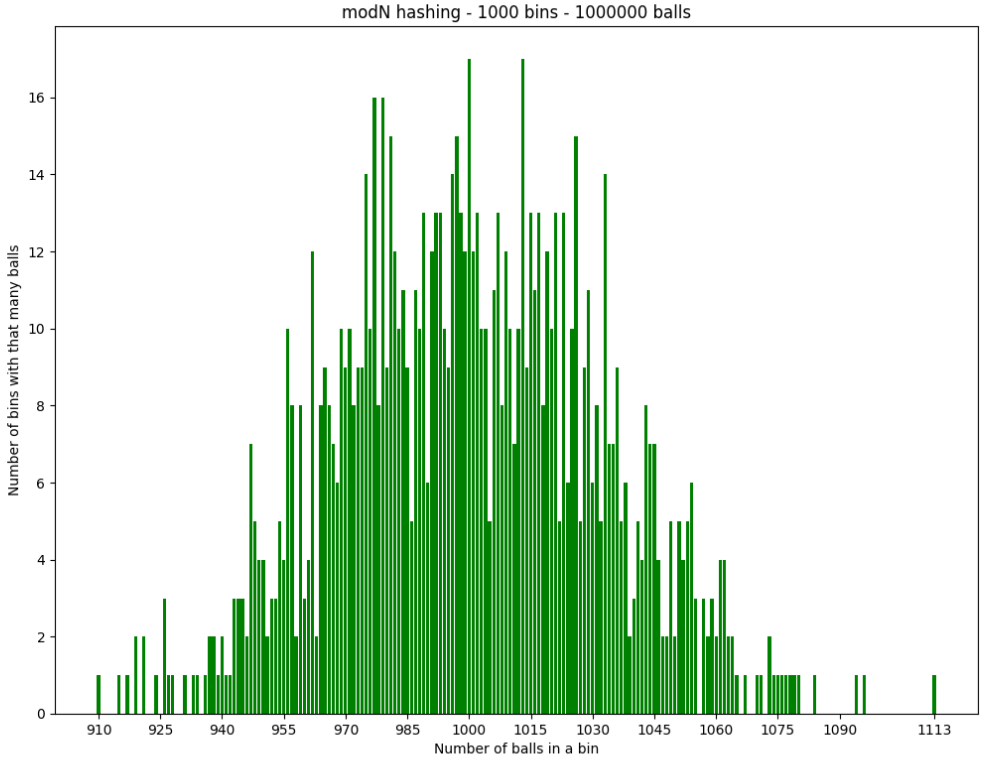


Figure 3.2: *Balls per bin distribution using **mod-N hashing**, for $N_s = 1000$ bins and $n = 10^6$ balls. $\frac{n}{N_s} = 1000$ balls per bin.*

A series of Monte Carlo simulations for various values of n, and a constant number of bins, $N_s = 1000$, indeed reveal that this intuition is probably justifiable. In Figure 3.2 and Figure 3.3, we can see how the number of balls per bin is distributed among the $N_s = 1000$ bins at the end of the simulation. The horizontal axis shows different values for the number of balls per bin, and the vertical axis shows the number of bins that contain that many balls.

In all simulations, the number of balls per bin, i.e. hash digests per shard, is distributed

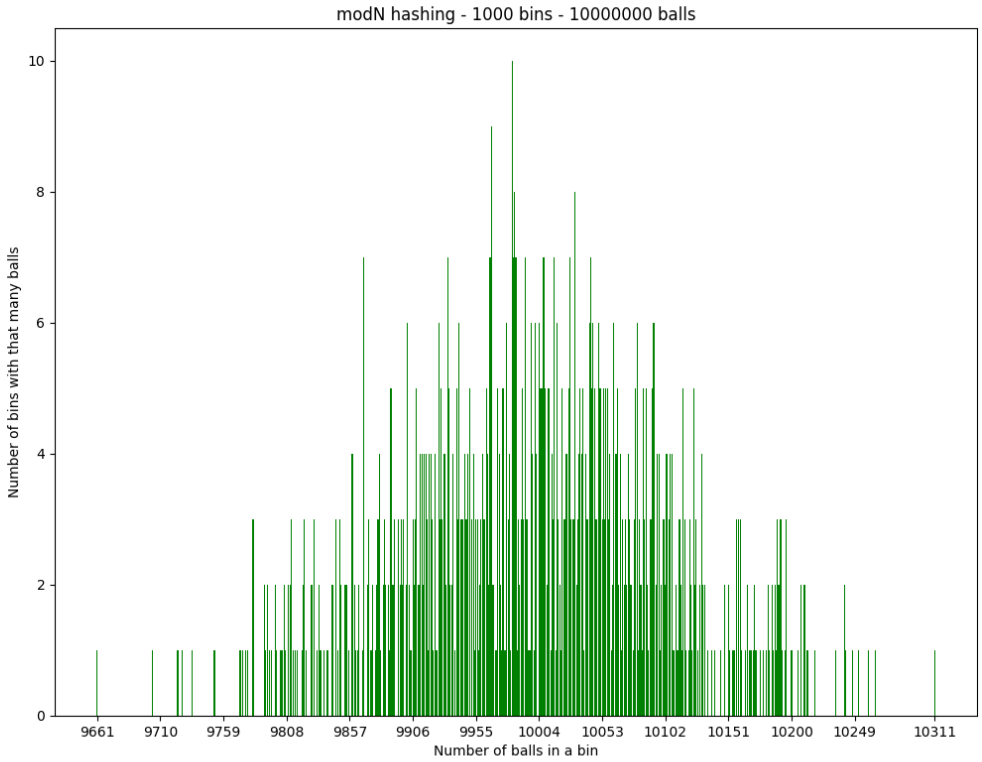


Figure 3.3: *Balls per bin distribution using **mod-N hashing**, for $N_s = 1000$ bins and $n = 10^7$ balls. $\frac{n}{N_s} = 10000$ balls per bin.*

around $\frac{n}{N_s}$.

Although mod-N hashing provides pretty decent load balancing as a sharding method, it has a major drawback: its capabilities regarding resharding. *Resharding* was explained in section 3.6, where we mentioned that the exact behavior of the system may vary depending on the sharding method. As we have already stated, in `MICAS`, the number of shards needs to be statically defined; in fact, dynamically modifying the number of shards, i.e. modifying it while the system is in use, is not supported at all. Having splitted the data into N_s shards, let us consider the case where the number of shards changes to a new value, N_s', which may be either lesser or greater than N_s, signifying the scaling-in or the scaling-out of the system, respectively. For most values of N_s', based on Equation 3.5, the vast majority of the hash digests that can be generated – hence, the corresponding objects that may be already stored in the system, too – are assigned to a different shard than they were assigned before. This results in an

almost complete rearrangement of the data among the nodes, whereas ideally it would be necessary to move only $\frac{N'_s - N_s}{N'_s}$ of the data when $N'_s > N_s$, or $\frac{N_s - N'_s}{N_s}$ of the data when $N'_s < N_s$, in order to end up with them being balanced among N'_s shards. The amount of data that ideally would need to be moved is lower as the value of $|N'_s - N_s|$ gets lower, i.e. when the number of nodes that the system is being scaled-out or scaled-in by is small, which is usually the case. The bad rearrangement is due to the nature of the modulo operation itself, since it has to be computed again for each hash digest using a different divisor.

Since MICAS resembles a large-scale, distributed, on-disk hash table, we can point out the similarity of this problem in terms of classic hash table implementations. In most hash table implementations, when the *load factor* increases, i.e. the number of elements assigned per bucket gets big, the time required for searching through the hash table degrades. For that reason, when the load factor reaches a threshold, the number of buckets is usually increased and all elements stored in it are relocated to the new buckets. Although this operation is expensive, it is justified by the fact that, normally, it will not be invoked frequently. However, the case is different for a distributed storage system, where such a solution might cause major availability issues, even if it happens infrequently – which might even not.

To cope with this difficulty, MICAS does not support resharding at all. The number of shards must be carefully provisioned at the time of the deployment of MICAS: since it cannot be modified later, the administrator of the system should configure the number of shards to the maximum value that it is expected to be needed. The actual number of shards in MICAS does not have to be tightly coupled to the number of the underlying cluster nodes, as we will soon see in detail, when its architecture is presented. When the number of shards is greater than the number of nodes, however, obviously some shards will be colocated. By sharding the keyspace right from the beginning into as many partitions as it is expected to ever be needed, we can handle the system's scaling-out merely by expanding the cluster of underlying physical nodes. As the system grows according to its needs, the shards, which densely populate the physical nodes at first, are gradually being scheduled and located apart from one another, along with their stored data, onto the new nodes that join the cluster. As a result, the system's underlying storage capacity is increased due to the addition of new nodes, while the number of shards remains the same. Scaling the system in is handled in a similar way: the Agents

that run on the physical nodes that are about to leave the cluster are first evicted by Kubernetes and relocated to nodes that are staying in the cluster after the scaling-in, along with the data they store. In this way, the system's underlying storage capacity can be decreased and its nodes can be released, while the number of shards still remains unmodified.

This kind of dynamic adaptiveness to different scale is what makes the system **elastic**, although only **to some extent**. The provided elasticity is upper bounded by the initial configuration of the static number of shards. Moreover, the cost of providing this limited elasticity can be too high when it is not actually required. For instance, a `MICAS` deployment that initially needs only as few as three nodes for its operation (which is the lowest number of nodes assuming the replication factor is $k = 3$ so as to retain resilience to $k - 1$ node failures, as we will see in subsection 3.8.3), but also has the potential of requiring the total storage capacity of as many as a couple of hundred nodes in the future, may push Kubernetes to its limits: Kubernetes supports no more than 100 Pods scheduled per node [184], but in this case this is exceeded if the storage capacity of each node is less than approximately 11 TiB. The next system that will be presented, `RICAS`, is anticipated to utterly address the problem of elasticity.

3.8.2 Replication

As it has been previously stated, the storage systems that we designed aim to be highly-available. To achieve that, there has to be a mechanism that allows objects to be created and read even when some of the nodes have failed; in other words, each system needs to be fault tolerant. Replication is a technique applied to ensure that objects are created at and read from multiple nodes in the system.

In the case of `MICAS` and mod-N hashing, the procedure is very simple. Since each shard can be thought of as a bin, the idea is to simply keep k replicas of every bin, where k is actually the replication factor. Whenever a "ball" (an object) should be placed in a bin, replicas of the ball are being placed in all replicas of the appropriate bin. Similarly, when a ball must be retrieved from a bin, it is known that a replica of that ball has been probably placed in all corresponding bin replicas, thus it may be retrieved from any of them. The logic is similar to *full node replication*, although, since as we will soon see in subsection 3.8.3, a number of shards may be colocated onto the

same physical node, calling it "*full shard replication*" would be more accurate.

The specifics of how this simple idea is incorporated into the architecture of `MICAS` become clear in the following subsections.

3.8.3 Architecture

As explained in section 3.7, MICAS consists of two tiers: the frontend and the backend.

The Proxies on the frontend can be deployed via Kubernetes using Pods controlled by either a `Deployment` or a `DaemonSet`, as explained in section 3.7. From now on, `Deployment` is our preferred Kubernetes API object for deploying the frontend, because it allows for scaling it out and in more easily. Regarding the deployment of the backend tier, in section 3.7 we mentioned that `StatefulSets` is probably the way to go, though we left the specifics out of the high-level description back then.

At this point, we are examining the deployment of the Agents of `MICAS` in detail. Every Agent is responsible for the data in a single shard. As outlined in subsection 3.8.2, the data of each shard are replicated, basically, by replicating the whole entity corresponding to the shard – the Agent. Put differently, multiple deployed Agents are responsible for exactly the same data – the data of a single shard – for every shard. Their precise number is k, equal to the replication factor, which is configurable. Each set (of size k) of Agents that are responsible for the same shard, is deployed via a separate Kubernetes `StatefulSet`.

Each one of the N_s shards is assigned a unique ID in the range $[0, N_s - 1]$, for the purpose of sharding using mod-N hashing. Every `StatefulSet` is also assigned that unique shard ID, and it is included on its name. This, in turn, is reflected both in its unique DNS address, internal to Kubernetes cluster and provided by the `KubeDNS` Kubernetes cluster Addon, and in the unique identifier, also for Kubernetes (in conjuction with the associated Kubernetes `Namespace`), allowing us to reach every underlying Pod that the `StatefulSet` controls, at any point, via Kubernetes `Endpoints`, also thanks to the `Headless Service`. Since every `StatefulSet` controls k Agent Pods, it may be handy to assign a descriptive name to these Pods as well. Thus, the Pods inside a `StatefulSet` are also assigned unique IDs, in the range $[0, k - 1]$, through their name. Ultimately, every Agent Pod is distinguished by two unique IDs in its

name, which ends up being in the form of "Agent-i-j", where i indicates the shard's ID and j indicates the "replica ID" of the Agent within the i-th shard. Along with the associated Kubernetes Namespace, this name, which, as we see, is deterministically assigned to every Agent Pod, can be utilized by both Proxies and other Agents, to contact an Agent, either by finding the corresponding Kubernetes Endpoints, or via the generated Kubernetes DNS address.

Figure 3.4 illustrates the architecture of MICAS. One may observe an arbitrary number of Proxy Pods being deployed via the "Frontend Deployment", a static number of deployed StatefulSets, one for each of the N_s shards, and each StatefulSet controlling k Agent Pods. The arrows indicate the flow of the data during a Create or a Read request, which we are about to go through next.

At this point, it is important to note that the actual scheduling of the Agent Pods across the physical nodes does matter. Agents that are scheduled by Kubernetes to execute on the same physical node, are all prone to the same node failures. Since all k Agents in a StatefulSet are responsible for the same portion of data stored in MICAS, Agent Pods controlled by the same StatefulSet should not be scheduled on the same node. Otherwise, a single node failure could cause multiple replicas of the same objects to be simultaneously unavailable. Effectively, this weakens the $k-1$ fault tolerance guarantee of the system, in the sense that $k-1$ Agent Pods may fail as a result of fewer than $k-1$ node failures. To address this problem we employ Kubernetes' affinity settings; in particular, Inter-Pod Anti-Affinity [180]. Every Agent Pod is assigned a special Kubernetes Label indicating the shard that it belongs to. These are used by Kubernetes' scheduler to factor out any nodes that are ineligible for an Agent Pod to run on: no two Agent Pods controlled by the same StatefulSet may be scheduled on the same physical node.

3.8.4 Cluster Phases, Failover & Recovery

Both MICAS and RICAS are scalable and self-healing systems. The vehicle to provide these important features is, of course, Kubernetes, which lies in the core of our systems' design, as it should be obvious by now. Kubernetes, not only schedules properly all the component entities of our systems when they are first deployed, but it is also re-

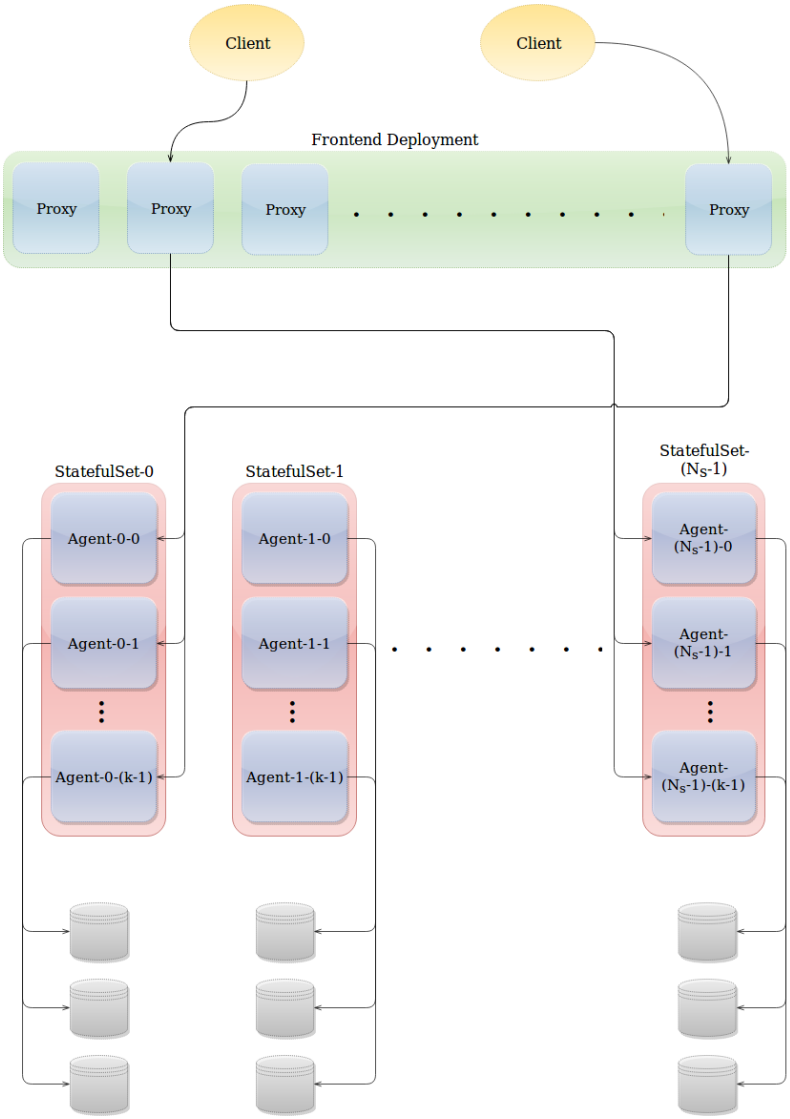


Figure 3.4: *The proposed architecture of* `MICAS`*, for a deployment consisting of an arbitrary number of Proxies and N_s shards, assuming the replication factor is k. The arrows indicate the flow of the data during a* `Create` *or a* `Read` *request.*

sponsible for tracking their status throughout their lifetime and making sure any failed entities are properly rescheduled and possibly relocated.

Upon starting its execution in its designated Pod, a `MICAS` Agent may find itself in one of two possible situations. Both of them are shown in Figure 3.5, and explained in the following paragraphs.

The first case is that the cluster has just been booted as a whole. That is, all Agent Pods have just been booted and all of them are in the **Bootstrapping phase**, and so is the "`MICAS` cluster" that, as we say, they form. When all Agents are up and ready

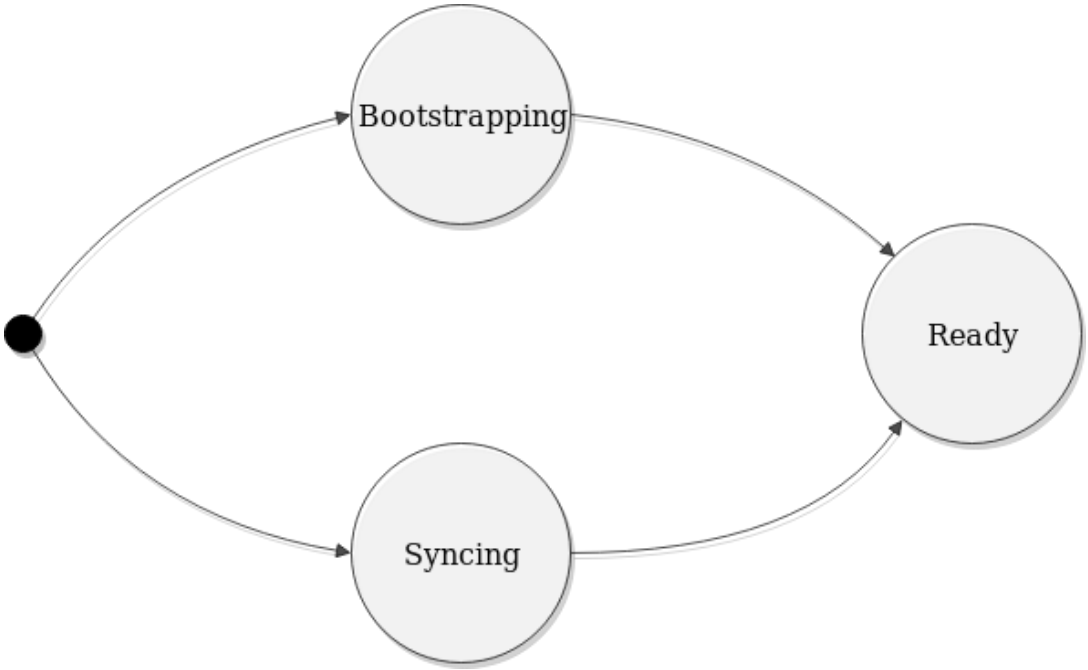


Figure 3.5: *State diagram for a* `MICAS` *Agent. Upon starting its execution, a* `MICAS` *Agent may find itself in either the* ***Bootstrapping*** *phase, or the* ***Syncing*** *phase. In both cases, it eventually reaches the* ***Ready*** *phase.*

to serve requests incoming from the Proxies, the cluster – and, of course, all Agents that comprise it – transit from the Bootstrapping phase to the **Ready phase**. In an environment prone to failures, such as ours, having multiple Agent Pods agree on when the Bootstrapping phase is over is not as simple as it may appear at first.

Again, Kubernetes comes to the rescue. Kubernetes offers *optimistic concurrency control* through its API objects by the means of the concept of *resource versions*, encompassed as a field in every API object's metadata. Basically, it is a string that identifies the internal version of an API object, which can be used by the clients to determine when objects have changed [183]. The resource version is included with any client's `PUT` operation so that Kubernetes API server can verify that there have not been any successful mutations to the resource during a read/modify/write cycle, by verifying that the current resource version matches the value specified in client's request. Currently, when etcd is used as the key-value store backend of the Kubernetes cluster, which is the case for our system as well, the resource version is backed by etcd's `modifiedIndex` field. This field is etcd's "sequencer", and can be thought of as a logical clock used by Kubernetes API server to order requests relevant to a specific key-value record.

The only way for a client to know the expected resource version is to have received it from the server in response to an earlier operation, typically a `GET`. This value must always be treated as opaque by clients and passed unmodified back to the API server. When two mutating operations on the same resource occur in parallel, only one of them actually succeeds. There is no way to know which one of them succeeds, but whichever it is, it also changes the resource version of the API object that it modifies, hence preventing the other mutating operation from being verified and applied.

This is ideal for our case, where Agents need to agree about being in the Bootstrapping phase or not. In `MICAS`, where the data of each shard are completely independent from those of the other shards, and the replication of the shards' data takes place by replicating whole Agent Pods via `StatefulSets`, each Agent only needs to coordinate with its peers in the same `StatefulSet`. No Agent ever needs to transfer data, or even contact for any other reason, another Agent living in a different `StatefulSet`. Therefore, there is no real reason to do so in the case of bootstrapping either: each Agent needs only care to agree on when the Bootstrapping phase ends with those that store the same data as itself does. To this end, when `MICAS` is deployed for the first time, i.e. when all the related `StatefulSets` are deployed among the rest of the entities, every `StatefulSet` includes a sentinel value, the `Bootstrapping Flag`, set in its metadata, specifically in the `Annotations` field. Upon starting its execution, every Agent checks whether the Bootstrapping flag in its associated `StatefulSet` is set or not. If it is, it waits until all the Agent Pods that are controlled by this `StatefulSet` are Ready, and then attempts to clear the flag by issuing the respective `PUT` operation to Kubernetes API server. After every failed `PUT` operation, the resource is re-read from the Kubernetes API server, using the appropriate `GET` operation, to make sure that the next attempt is issued against the latest version of it. These attempts take place continually until the Bootstrapping flag is cleared. When this happens, we say that the Agents that represent this shard have left the Bootstrapping phase; when all deployed Agents have left the Bootstrapping phase, so does the `MICAS` cluster, implicitly.

Node failures while being in the Bootstrapping phase do not cause problems to any Agent in the `StatefulSet`. We will examine it from both the perspective of the failing Agent, and from this of a non-failing Agent, for the sake of completeness. From the perspective of the failing Agent, it is simply rescheduled, possibly to another eligible node, and restarted. If the rest of the Agents in its `StatefulSet` are still in the Boot-

strapping phase, they should be waiting for the failing Agent, possible among others, too. Thus, he simply enters this phase again and follows the bootstrapping procedure as described above. If the rest of the Agents in its `StatefulSet` have moved passed the Bootstrapping phase, the failing Agent follows the procedure described below, as per Figure 3.5. The perspective of a non-failing Agent in the same `StatefulSet` largely depends on the exact timing of the failure. However, similarly, it does not cause troubles for the cluster. The failure may go unnoticed, e.g. if it happens during the attempts to clear the Bootstrapping flag, in which case the non-failing Agents simply proceed to Ready phase as described above. Otherwise, if it happens before those attempts begin, the failure may be noticed; then, the non-failing Agents simply wait until the failover completes and the failed Agent is restarted, as also described above.

The second case is that the cluster has not just been deployed. That is, an Agent may be starting to execute just now, but it observes that the Bootstrapping flag of its `StatefulSet` is not set, which means that its peers are not in the Bootstrapping phase, and probably neither is the cluster. In this case, the Agent figures out that it is probably just itself restarting from a previous failure, possibly after having been rescheduled to a different physical node by Kubernetes. Therefore, it enters the **Syncing phase** and begins its initial data synchronization procedure, as mentioned in subsection 3.7.2, which heavily relies on **rsync** at its core.

As it has been also mentioned in subsection 3.7.2, Agent Pods consist of two Docker containers each, rather than a single one. The first contains a process that implements our own `MICAS` Agent's logic, and the second contains an always-listening rsync daemon server, as shown in Figure 3.6. The rsync daemon is deployed in a separate container instead of being deployed as another process within the same container as the Agent, so as to leverage Docker daemon's features for automatically restarting it and independently managing it whenever required throughout its lifetime on the node. The job of the rsync daemon is simple: to export a single module, which exposes the root directory where all Agent's data are stored, and to serve any subsequent rsync client connection directed to that module.

The initial data synchronization of a `MICAS` Agent is a quite straightforward procedure. Upon entering the Syncing phase, the Agent simply forks off one rsync client process for each one of the rsync daemons in the Agent Pods of the `StatefulSet` that itself

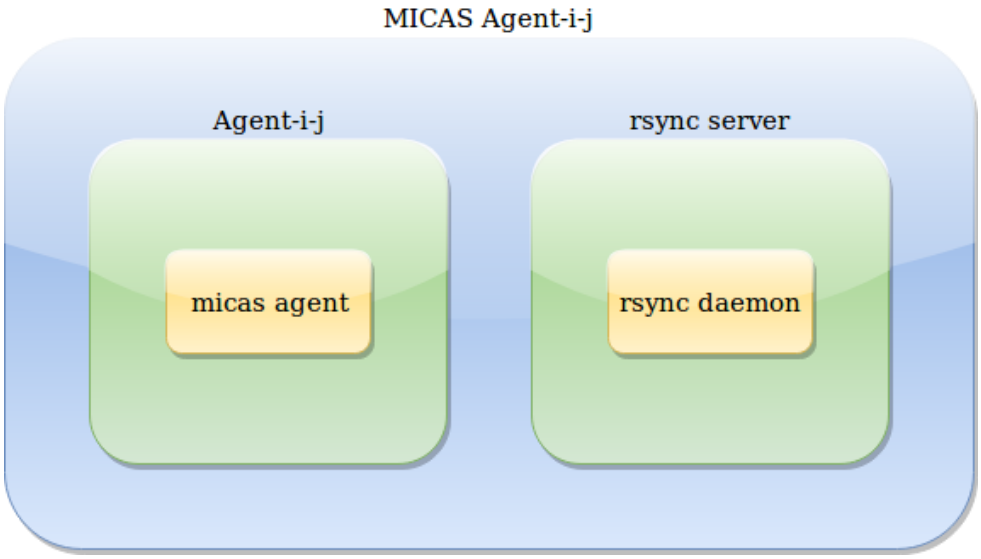


Figure 3.6: `MICAS` *Agent Pod consists of two containers, one for the actual logic of the Agent, and another one for an always-listening rsync daemon, which serves requests from Agents of the same* `StatefulSet`*, to assist them to their initial data synchronization.*

belongs to. If no old data are present, then all – or almost all – of them should have been transferred to the syncing Agent by the time the first rsync client process finishes its execution. Thanks to rsync's efficient algorithm, any subsequent rsync client processes that connect to the rest of the Agents in the `StatefulSet` to synchronize the same data, only transfer the portions of data that were not already pulled earlier during the same Syncing phase. Such portions may exist due to the per-object consistency level being configured at a value that is lower than the replication factor for some objects, a subject which will be discussed on the next subsection. Obviously, rsync's algorithm works wonders in cases that most of the Agent's data are already there upon restarting. In this case, only the objects that were created since the Agent failed will be transferred – to be precise, any older data missing because of the aforementioned consistency level configuration will be transferred as well.

It is important to note that during the initial data synchronization, except from synchronizing their data with their peer-Agents via rsync, all Agents also make an effort to serve any incoming request (incoming from the Proxies, emanating from the clients). Thanks to immutability, content-addressability and, of course, the lack of support for update operations, we rest assured that the set of objects that are being transferred in the context of the initial data synchronization and the set of objects that are being

created due to Proxy requests, are disjoint. This is an important observation for making sure that problems related to race conditions while writing objects are completely avoided. Even in the case that rsync schedules the transfer of an object from another Agent right before a `Create` request for that object arrives from a Proxy, considering that the objects are immutable and content-addressable and that both rsync and our storage engine ensure that new objects are created atomically, no data inconsistencies can ever appear.

Node failures during the Syncing phase do not affect any Agent in a `StatefulSet` other than the one that runs on the failing node. As usually, once the failure occurs, Kubernetes reschedules the affected Agent, possibly relocating it to another eligible node. Then, the restarted Agent simply enters the Syncing phase again and begins its initial data synchronization normally. There is still possibility that some portion of the data already exists, thus taking advantage of rsync algorithm's efficiency, as it was previously explained, is still possible.

3.8.5 Data Flow

Now that we know about the architecture of `MICAS`, we may have a closer look on what exactly happens when `Create` and `Read` requests arrive.

As explained earlier, the frontend is responsible for receiving the requests from clients and applying the sharding technique of choice, mod-N hashing in this case. On object creation, a Proxy receives an incoming blob and uses it as an input to the predefined hash function to generate a hash digest, the new object's key. Next, based on Equation 3.5, the Proxy calculates the unique ID of the shard that this object belongs to. Knowing which `StatefulSet`'s Agent Pods should the object be stored at, it sends the blob along with its hash digest to all k of them using their internally exposed REST or RPC API. It then blocks, waiting to accumulate their responses. To avoid long delays or complete unavailability towards the clients due to single node failures, Proxies set a time-out for their communication with the Agents.

On the backend, when a `Create` request arrives from the frontend, the procedure is pretty straightforward. All an Agent has to do is durably persist the blob using the underlying storage engine and the hash digest as the key, where both the blob and its

key should be contained in the request. The specifics may vary among various storage engines. In our case, no sophisticated storage engine is used; we use a dummy one, merely to ensure the correctness of our design under certain conditions. This comes at a huge performance cost, though; accessing the disk, writing data to it and, most of all, making sure that the writes are actually **durable**, is an extremely slow procedure if designed and implemented naively – as in the case of both `MICAS` and `RICAS`. Therefore, this is expected to be a major bottleneck in our system's performance. More on the design and implementation of our naive storage engine will be discussed in section 4.1. The Agent responds to the Proxy that the object has been successfully and durably created locally, otherwise it reports any error that might have occurred.

The Proxy processes the responses one-by-one as they arrive from the Agents. By default, the Proxy considers an object created only if all k Agents report that they successfully stored it locally; on the first report of an error received from an Agent, the Proxy considers the `Create` request failed, and replies to the client properly. The Proxy considers the request failed, too, and replies to the client accordingly, if any number of the k Agents has not responded at all after the duration specified by the time-out passes. In other words, by default, the creation of an object is considered failed unless all expected k replicas have been successfully created. This design choice allows us to argue that our system is both strongly consistent and $k - 1$ fault tolerant as far as the creation of new objects is concerned: once a client receives a successful response for a `Create` request, `MICAS` guarantees immediate visibility of the associated object to all clients. This sort of guarantee is sometimes called **read-after-create consistency**. Once the system responds that an object has been successfully created, it is guaranteed that the system tolerates as many as $k - 1$ Agent failures within the same shard – or more if the failures occur across different shards – without losing the ability to access that object: under read-after-create consistency it is certain that there are k durably persisted replicas of that object, hence, k Agents ready to serve any subsequent `Read` request for it.

Even though read-after-create consistency is often a useful guarantee, it may not be a strong requisite for some applications. Sometimes, applications prefer low latency over consistency. Waiting until all k object replicas are durably persisted on all k Agents of a shard, can obviously introduce delays, since the Proxy will not respond to the client until the "slowest" Agent in the shard reports that it successfully stored a replica of the

object locally. For that reason, as a feature, the client is allowed to configure the number of object replicas that should definitely be persisted durably before the system responds successfully. This is possible by simply including an integer, c, in the range $[1,\ k]$ in the `Create` request of the object. The value of c may be subject of some policy or the business logic of an application; this is why it should be configurable independently for different clients, and even for `Create` requests about different objects issued by the same client. The integer is processed by the Proxy that receives the `Create` request, instructing it to report success to the client even if it has been confirmed for only c out of k replicas that they have been locally stored by an Agent in a durable manner within the allowed duration determined by the time-out. This is, essentially, a kind of **adjustable read-after-create consistency level** at **per-object granularity**, in favor of low latency as perceived by the clients. It is important to note that, although from a client's perspective the latency may indeed be lower, the Proxy still has to deal with Agents' delayed responses for at most as long as the specified time-out, hence, the amount of underlying resources needed by `MICAS` is not really reduced in any way by this feature.

The ability to adjust the consistency level does not come without downsides. If c in an object's `Create` request is set to a value below k, the client cannot be totally sure about this object's availability. Let us consider, for instance, that a client configures $c = k - 1$ in a `Create` request, and that the Proxy reports to the client that the creation of the object was successful, having, indeed, exactly $c = k - 1$ replicas of the object persisted durably on exactly $k - 1$ Agents. Hypothetically, the k-th replica was not successfully stored; this might have happened for any reason, for example the k-th Agent may have been experiencing some kind of failure. In the – admittedly unlikely, yet possible – scenario that the $k - 1$ Agents that successfully stored the replicas of the object all fail by the time the k-th Agent is restarted, but before it completes its initial data synchronization, the object at hand will not be available to the client, albeit the client has been informed that the object was created successfully. This would happen because the Proxy associated with that object's `Create` request only waited for c confirmations instead of k. In other words, configuring $c < k$ for some objects weakens the guarantee of fault tolerance from $k - 1$ to $c - 1$ in the worst case, for these objects. Of course, once the failing Agents are restarted, the object would become available again.

The flow of the data is quite similar during a `Read` request. First, the client request arrives at the frontend. It contains the unique key assigned to the requested object at the time of its creation. The Proxy, based on the mod-N hashing technique for sharding and using Equation 3.5, calculates on which shard should the object have been stored, if it is stored in `MICAS` at all. Once the shard, i.e. the `StatefulSet` for the case of `MICAS`, is decided, the Proxy reaches its k Agents through their internally exposed API. It issues a `Read` request to all of them simultaneously, including the object's key in it, as it was received by the client, and then it blocks waiting for their responses.

When an Agent receives a `Read` request from a Proxy, it extracts the requested object's key, and queries its local storage engine to retrieve the stored blob, if it exists at all. If it does exist, it is included in the Agent's response to the Proxy. Otherwise, the Agent reports back to the Proxy that no object with the given key is stored.

The Proxy receives Agents' responses and processes them one-by-one. If an Agent's response is negative, i.e. there is no object associated with the particular key stored locally to that Agent, then the Proxy keeps waiting for the responses of the rest of the Agents. If all k Agents' responses are negative, then the Proxy responds to the client that the requested object does not exist. On the first positive response that it receives, the Proxy immediately responds to the client, including the retrieved blob in the response.

It is obvious that if more than one Agents have the requested object stored locally, all of them will respond to the Proxy with the relevant data. As the responses are sent over the network, it is also obvious that network bandwidth is being wasted on the transfer of the same data from multiple Agents to the same Proxy. In fact, if all k Agents have a replica of the object, the allocated network bandwidth is k times more than the minimum bandwidth that would actually suffice to handle the associated client `Read` request. This is a design choice that had to be made. Our rationale is that, in its simplicity, one of our systems' most typical workloads is expected to consist of many `Read` requests for the stored immutable objects, and these requests should be handled as fast as possible. Forwarding clients' `Read` requests to the Agents one-by-one until one of them finally responds with the blob, would certainly avoid wasting bandwidth altogether. However, it could potentially cause long delays to a client in cases that the requested object is stored to only some of the k `MICAS` agents. The worst case scenario

is that a number of the k Agents exhaust the time-out duration set for the communication between Proxies and Agents. By querying all k Agents simultaneously for the object, we can be sure that a client's `Read` request for an existing object is handled as soon as the "quickest" Agent pushes the blob to the Proxy, although at the cost of having all of them pushing it. In the future, as an optimization, the Proxy could be implemented to forward a client's `Read` request only to $k' < k$ Agents, in iterations, until either the object is found or all k Agents have been reached. Thus, the network bandwidth being wasted would be upper bounded to k' times more than the bandwidth that is actually needed, again, of course, at the cost of slightly increased latency as perceived by the client.

3.9 RICAS

`RICAS` stands for Ring-based Immutable Content-Addressable Storage. It is the second of the two systems presented in this thesis, and the result of our attempts to improve `MICAS` in terms of elasticity, without sacrificing any major part of the rest of its functionality.

3.9.1 Sharding using Consistent Hashing

`RICAS` employs an alternative sharding technique, somewhat more complex than `MICAS`' mod-N hashing. It is commonly known as **consistent hashing** algorithm, or **consistent hashing ring** method, and it is going to be thoroughly explained in this section. Its conception is attributed to Karger et al. in 1997 and it was originally presented in [21].

The original motivation for consistent hashing has been Web caching. That refers to an efficient way to implement a large-scale, multi-node caching system for web content that acts as a cache shared by multiple users. As an extra constraint, the number of the caching server nodes may inevitably change over time instead of being static, either due to need for scaling the system out or in, or due to some of its nodes failing. As we will examine, this situation is quite identical to ours, except that in our situation we talk about Agent Pods rather than physical nodes.

As explained in subsection 3.8.1, `MICAS` requires configuring a static number of shards, N_s, throughout the lifetime of the deployment, and relies upon Kubernetes' scheduler for properly assigning these shards to the nodes of the underlying Kubernetes cluster when the system needs to scale out or in. On this basis, `MICAS`' sharding technique was juxtaposed with a classic hash table implementation, except that rehashing and relocating the objects among the shards was prohibitively expensive in terms of availability, thus unsupported altogether. The primary aim of consistent hashing is to retain as many objects as possible to the same shard even as the number of shards changes, i.e. to minimize the number of objects that need to be moved during such a change. A key idea to achieve this, is to use **keyspace ranges** instead of the modulo-based logic of mod-N hashing for sharding, in a way that we intend to shed some light on very soon.

We start off by choosing a N_h-bit hash function, h, taking into account the calculations of section 3.5. The hash function is used during each object's creation to generate a hash digest given the content of the incoming blob as an input. As explained in section 3.4, the N_h bit long output is the content address of the new object, hence also its key. Another key idea behind consistent hashing, and a major difference compared to mod-N hashing, is that this time the "nodes", which are the Agents in our case, must also be assigned a N_h-bit hash digest, probably using the same hash function, h. This hash digest is generated using each Agent's unique name as input to the hash function [27].

To understand how objects are assigned to shards, let us consider the whole keyspace on an axis, as shown in Figure 3.7. Every possible output value that can be generated by the application of h on an input, is represented as a N_h-bit integer on this axis. Therefore, the values that are included in the axis are all integers in the range $[0,\ 2^{N_h} - 1]$.

Next, let us imagine that a total of N_s Agents have already been assigned a hash digest in this range, based on their unique name, and can therefore be conceptually "placed" on that axis. Figure 3.8 depicts the axis marked with the N_s hash digests, S_i for the i-th Agent, where $i \in \{0, \ldots, N_s - 1\}$.

[27] More on Agents' names are discussed later in this subsection, as well as in subsection 3.9.3; until then we only need to mind that every Agent is indeed given a name that is unique among the names of its peer-Agents in the deployment of `RICAS`.

Figure 3.7: *An axis on which all possible outputs of the N_h-bit hash function can be conceptually "placed", thus representing the whole keyspace.*

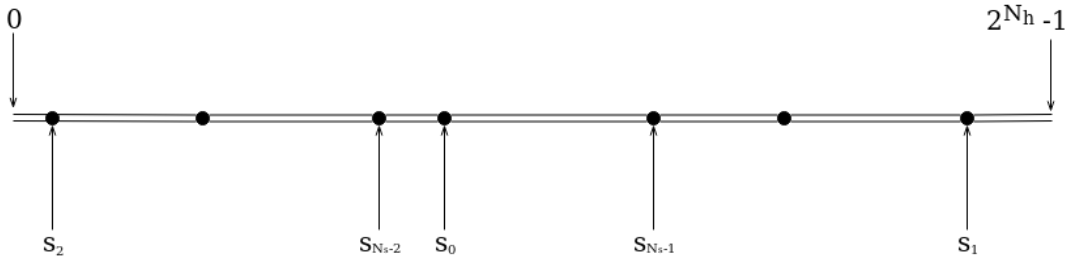


Figure 3.8: *The axis of Figure 3.7, annotated with the hash digests of the N_s Agents. S_i represents the i-th Agent, were $i \in \{0, \ldots, N_s - 1\}$.*

Given a blob, b, that hashes to $h_b = h(b)$, we scan the axis to the right of h_b until we find an integer S_j to which the name of one of the Agents hashes. Blob b is assigned to the j-th Agent, the one corresponding to hash digest S_j. An example of this procedure is illustrated in Figure 3.9, where S_1 and S_{N_s-1} are designated as the Agents responsible for storing the objects with keys $h_{b_1} = h(b_1)$ and $h_{b_2} = h(b_2)$, respectively.

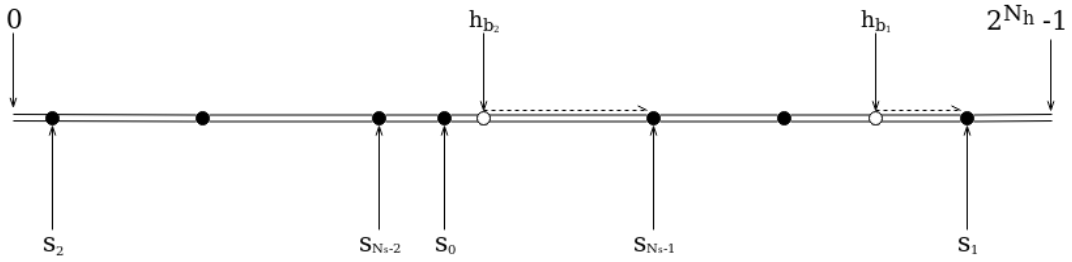


Figure 3.9: *The axis of Figure 3.8, additionally annotated to include object keys $h_{b_1} = h(b_1)$ and $h_{b_2} = h(b_2)$. One can see how h_{b_1} is assigned to Agent S_1 and h_{b_2} is assigned to Agent S_{N_s-1}.*

In other words, each Agent that is placed onto the axis represents a shard in the keyspace of the storage system. This shard includes every object that is assigned a key in the range between the hash digest of the Agent laid to its left on the axis, and its own hash digest. Figure 3.10 is an attempt to depict the shards on the axis using different colors for each Agent and its associated keyspace range.

Obviously, a number of object keys may not have any Agent hash digest to their right.

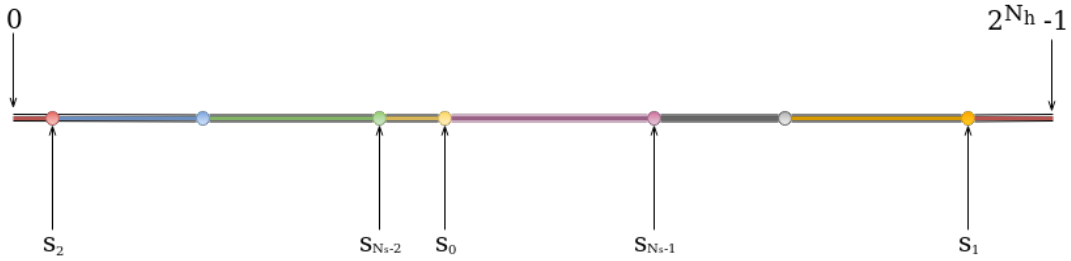


Figure 3.10: *The axis of Figure 3.9, this time using a unique color for the key range that each Agent is responsible of, in an attempt to show the different shards of the keyspace. One can see how we wrap around the axis to include the object keys in the rightmost shard to the leftmost Agent.*

As shown in Figure 3.10, in this case we wrap around the axis, assigning these objects to the leftmost Agent on the axis. Consequently, it might be conceptually more accurate to think of spreading the keyspace on a circle instead of an axis; hence the consistent hashing "ring" terminology, which is where the name of `RICAS` is derived from, too. In the ring representation, in order to find which Agent is responsible for a given key, we scan the ring in a clockwise direction until an Agent's hash digest is reached.

Figure 3.11 is the same as Figure 3.10, with the exception that the keyspace is shown as a ring instead of a straight line.

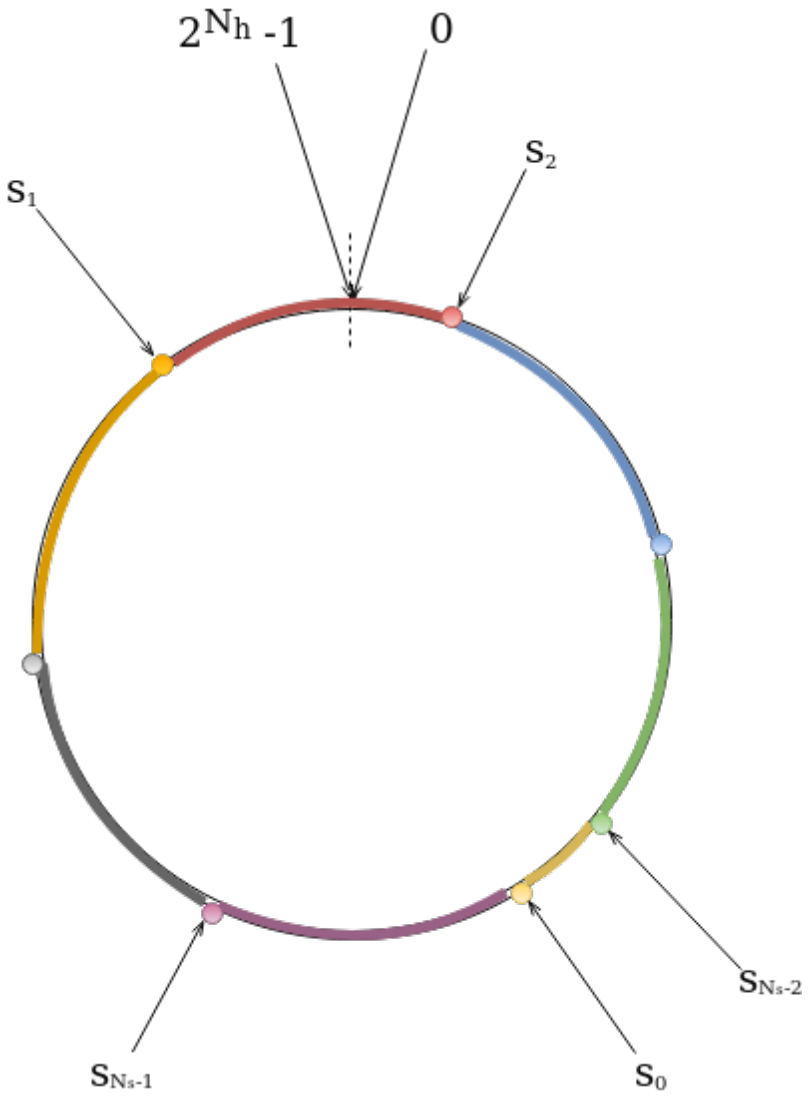


Figure 3.11: *The keyspace shown in axis of Figure 3.10 can be spread on a circle instead of a straight line, forming a visual representation of the "**consistent hashing ring**".*

Similarly, in Figure 3.12 we can see the same example of the sharding procedure as in Figure 3.9, using the ring representation instead of the axis.

The greatest benefit of employing consistent hashing for sharding is the provided **elasticity** in cases where *resharding* is required. Resharding was discussed in section 3.6, and then was examined in the context of `MICAS` in subsection 3.8.1. Contrary to `MICAS`, the number of shards in `RICAS` does not have to be statically defined; it can be adjusted both upwards and downwards depending on the scalability needs of the system each time.

Let us consider the keyspace splitted into N_s shards; to visualize it, one can imagine

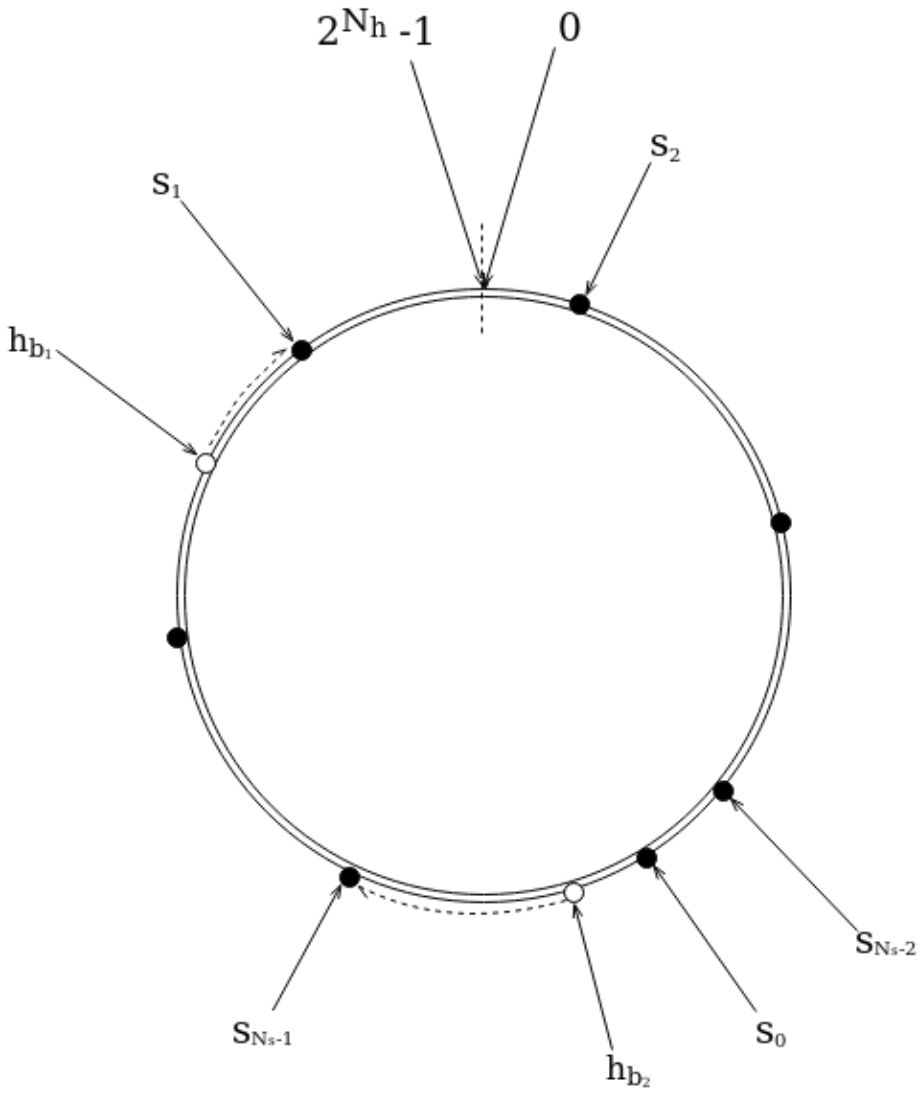


Figure 3.12: *The sharding procedure, as shown in Figure 3.9, can also be visualized on the consistent hashing ring. In order to find which Agent is responsible for a given key, we have to scan the ring in a clockwise direction until an Agent's hash digest is reached.*

that the ring should look as in the Figures that have been presented so far. We need to scale `RICAS` out by one node, i.e. one more Agent, thus one extra shard. Since each Agent is responsible for a single shard and is represented by a single node in the ring, the terms "Agent", "shard" and "*ring node*", henceforth, may sometimes be used interchangeably in the context of the description of a consistent hashing ring to denote either the associated segment of the keyspace or the related Agent entity that is responsible for it. In contrast to `MICAS` and mod-N hashing, now, we may simply let a new Agent join the cluster by joining the ring, as usual: its unique name is hashed and placed onto the ring according to the value of its hash digest. Therefore, a new seg-

ment is created on the ring, which corresponds to the shard that the new Agent should from now on be responsible for. The new shard extends from the hash digest of the "previous" Agent in a leftwards (anti-clockwise) direction of the joining Agent, until the hash digest of the latter. This segment of the keyspace must had been previously assigned to the Agent that now is the "next" one in a rightwards (clockwise) direction of the joining Agent.

Figure 3.13 is based on the case shown in Figure 3.11, where there are N_s shards formed by placing the hash digests of the corresponding Agents, $S_i, i \in \{0, \ldots, N_s - 1\}$. We assume that the system needs to be scaled out by one Agent. It depicts the segment on the ring that corresponds to the new shard that is being created and assigned to the new Agent, by placing onto the ring the hash digest of the latter, S_{N_s}. It is evident that the objects assigned to the shard of the new Agent S_{N_s} must had been previously assigned to Agent S_1.

The rationale of the procedure of removing a shard is pretty much the same as the one described above, when we examined the case of adding a shard. When `RICAS` needs to be scaled in, say by one Agent, we always remove the Agent – hence also the shard – that last joined the cluster. The objects of this shard, those that lie in the keyspace range delimited by the hash digest of the leaving Agent and that of the "previous" Agent in an anti-clockwise direction of it, are assigned to the "next" Agent placed in a clockwise direction of the leaving one, as they should be before the leaving Agent joined the cluster. For example, in Figure 3.13, if Agent S_{N_s} had to be removed, all objects designated to it should now be assigned to S_1.

The great advantage of consistent hashing, which essentially justifies it being a massively scalable and elastic solution to the problem of sharding, is hidden in the concept of ring segments and the way they are splitted or merged with one another whenever the system needs to scale. Details, as well as formal explanations of the concepts, can be found in [21]. Here, we mention just two of its relevant properties, which satisfy the intuition from the description that has been given so far. First, a paraphrase of its definition in [21], there is a "**smoothness**" property: *when an Agent is added to or removed from the cluster, the expected fraction of objects that must be moved to a new Agent is the minimum needed to maintain a balanced load across the Agents*. Put differently, this property implies that smooth changes regarding the size of the cluster of

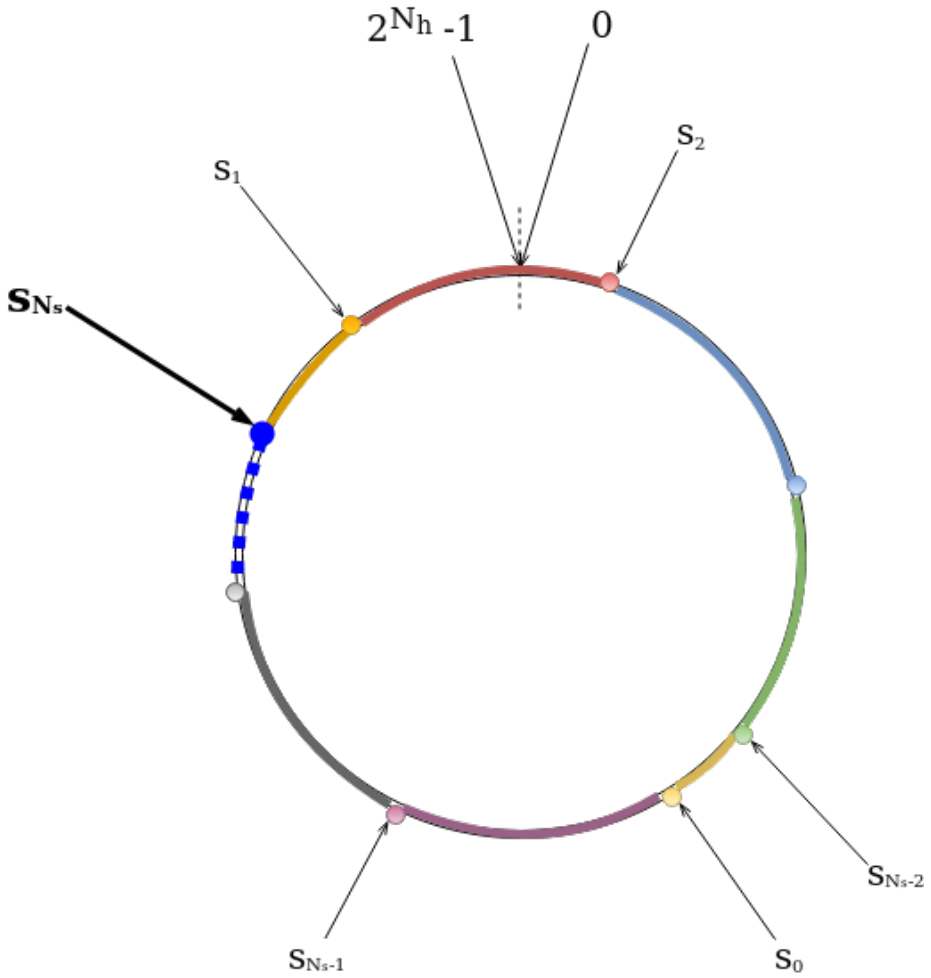


Figure 3.13: *The consistent hashing ring of Figure 3.11 is being scaled out by one Agent. A new shard, S_{N_s}, is created and assigned to the new Agent by placing the hash digest of the latter onto the ring. The objects assigned to the shard of the new Agent S_{N_s} must had been previously assigned to S_1.The situation is similar (but reverse) in the case of scaling the consistent hashing ring in, from $N_s + 1$ Agents back to N_s Agents, by removing the newest Agent S_{N_s}. This time, all objects that belonged to S_{N_s} are assigned to S_1.*

Agents are matched by a smooth evolution in the location of the stored objects. Second, consistent hashing ensures **monotonicity** (again, a paraphrazed definition based on [21]): *when the total number of Agents changes, objects may move from the directly affected ring nodes towards the unaffected (or affected only indirectly) ones, but never the other way around.* For instance, when new Agents are added to the cluster, stored data may move only from the old Agents to the new ones, but never towards other old Agents. Similarly, when Agents are removed from the cluster, stored data may move only from the leaving Agents to the ones that are staying, but no data would ever move *from* an Agent that is staying after the scale-in. "This reflects one intuition about con-

sistency: when the set of usable buckets changes, items should only move if necessary to preserve an even distribution" [21].

However, consistent hashing has a downside which is not present in mod-N hashing. Apparently, some of the significant load balancing benefits of the latter are sacrificed for the sake of perfect horizontal scalability and elasticity. Ideally, assuming that the analysis of section 3.5 has been taken into account, the expected load of the N_s Agents is a $\frac{1}{N_s}$ fraction of the objects. In practice, however, there is a non-trivial variance to the load of each Agent. To visualize that, if one picks N_s random points on the circle, it is highly unlikely that the circle will be partioned perfectly into segments of equal size. Put another way, some of the shards may be assigned a much larger range of the keyspace than others; hence, some Agents may be responsible for a much larger fraction of the objects than others. This observation is already reflected in the Figures of the ring that have been presented in this subsection so far.

As it has already been explained in subsection 3.8.1 about mod-N hashing, load balancing is modeled after the classic "balls and bins" problem in probability theory. Every shard can be modeled as a *bin*, and every object that needs to be stored in the system can be modeled as a *ball* that has to be put into some bin. To examine the quality of the load balancing of the objects among the shards, we have to examine the distribution of the balls among the available bins. A series of Monte Carlo simulations for a constant number of bins, $N_s = 1000$, and various values of n, the number of balls, confirms our intuition about the disparity in the distribution of the load. In Figure 3.14 and Figure 3.15, we can see how the number of balls per bin is distributed among the $N_s = 1000$ bins at the end of the simulations. The horizontal axis shows different values for the number of balls per bin, and the vertical axis shows the number of bins that contain that many balls.

Ideally, in all simulations, the number of balls per bin, which models the number of object keys per shard, should be distributed around $\frac{n}{N_s}$. However, it obviously is not. It is evident that some bins are assigned a number of balls much smaller than $\frac{n}{N_s}$, in the order of $0.46\frac{n}{N_s}$ or less, while other bins gather a quite large amount of balls, even in the order of $6.7\frac{n}{N_s}$ in some cases.

Fortunately, there is a straightforward way to decrease this variance: by mapping each

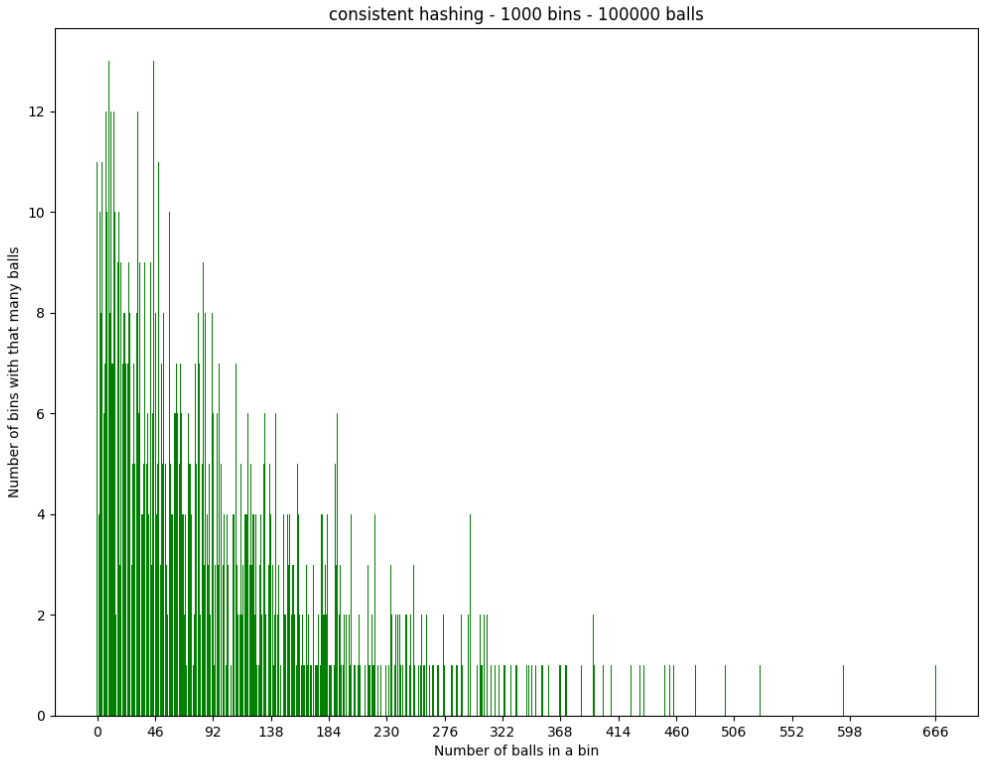


Figure 3.14: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins and $n = 10^5$ balls. $\frac{n}{N_s} = 100$ balls per bin.*

Agent multiple times on the ring. Having each Agent mapped V times on the ring, the latter is partitioned into $V \cdot N_s$ segments instead of only N_s. The $V \cdot N_s$ segments on a consistent hashing ring, created by mapping a ring node multiple times on it, may henceforth be also referred to as **virtual nodes** or **vnodes**. Based on the same reasoning as above, the greater the number of random points we pick on the circle, the higher is the probability that the subranges of the keyspace which correspond to the segments that are being created, are roughly of the same size. Therefore, the expected load of each virtual node is about $\frac{1}{V \cdot N_s}$ on average. Even in cases that the load of some of the segments still is asymmetrically large or small compared to this average, such outliers for each Agent are expected to largely cancel each other out, hence providing a decent load balancing among the ring nodes overall.

How can an Agent be mapped multiple times on a consistent hashing ring, though? There are various ways to achieve that. A simple way to map every Agent V times

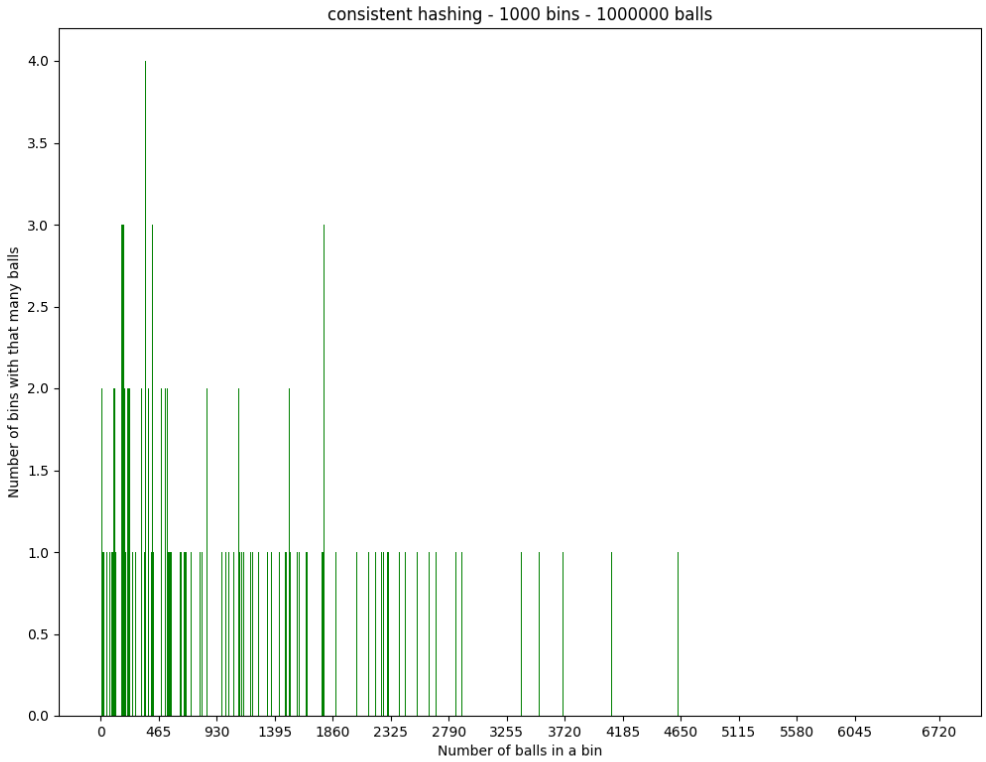


Figure 3.15: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins and $n = 10^6$ balls. $\frac{n}{N_s} = 1000$ balls per bin.*

on the ring, is to use V different N_h-bit hash functions. As long as a hash function fulfills the requirements posed in section 3.5 and its output is indeed N_h bit long, so that all generated hash digests always fall within the range $[0,\ 2^{N_h} - 1]$, it is an eligible candidate for such use. In the case of `RICAS`, a different approach is put into use, though. Knowing that every Agent can be uniquely identified, it is possible to deterministically generate a unique sequence of V identifications for its virtual nodes based on the original one. The same initial hash function, h, can thereby be used V times to produce the V hash digests of every Agent, thus avoiding the need to choose multiple hash functions, eligible as per section 3.5.

Figure 3.16 illustrates a consistent hashing ring of five ring nodes ($N_s = 5$), with two virtual nodes each ($V = 2$). The total number of segments is $N_s \cdot V = 10$, and each segment is represented by the hash digest of the corresponding virtual node, $S_{i,j}$, where $i \in \{0, \ldots, N_s - 1\}$ and $j \in \{0, \ldots, V - 1\}$.

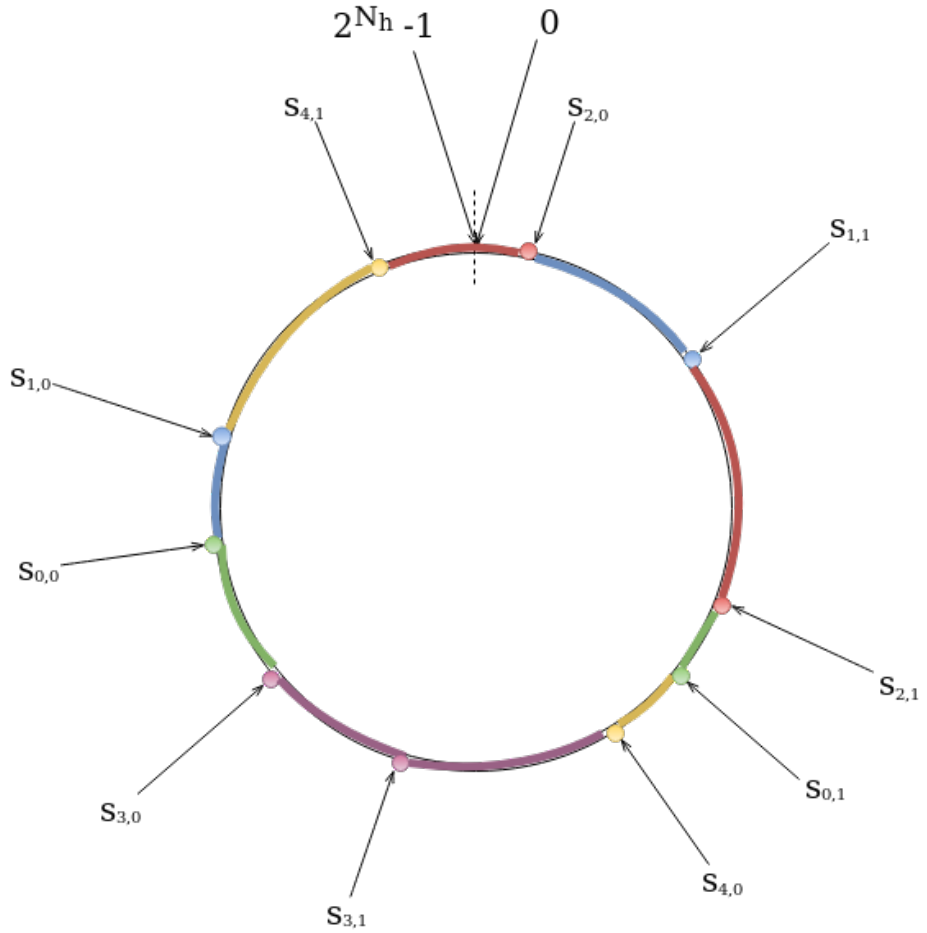


Figure 3.16: *A consistent hashing ring of five ring nodes ($N_s = 5$) with two virtual nodes each ($V = 2$). Each segment is represented by the hash digest of the corresponding virtual node, $S_{i,j}$, where $i \in \{0, \ldots, N_s - 1\}$ and $j \in \{0, \ldots, V - 1\}$.*

The logic for the assignment of the objects to the appropriate Agents now also has to take into account the presence of the virtual nodes per ring node, but otherwise it is the same as before. First, the hash digest of the object is placed on the ring. Then, moving in a clockwise direction, we scan the ring until we meet the next virtual node, $S_{i,j}$, that has been hashed onto it. This virtual node is considered to be responsible for that object. Of course, since the virtual node $S_{i,j}$ really is only a portion of the shard held by the i-th Agent, it is the i-th Agent that is actually responsible to store the new object. To visualize that, in Figure 3.17 we can observe how blobs b_1 and b_2, with hash digests respectively $h_{b_1} = h(b_1)$ and $h_{b_2} = h(b_2)$, are assigned to S_1 and S_3, respectively. Scanning the ring in a rightwards direction starting from h_{b_1}, the virtual node $S_{1,0}$ is first met, which is the first virtual node ($j = 0$) of the second ring node ($i = 1$) in the cluster. Similarly, starting from h_{b_2}, the virtual node $S_{3,1}$

is first reached, which is the second virtual node ($j = 1$) of the fourth ring node ($i = 3$) in the cluster.

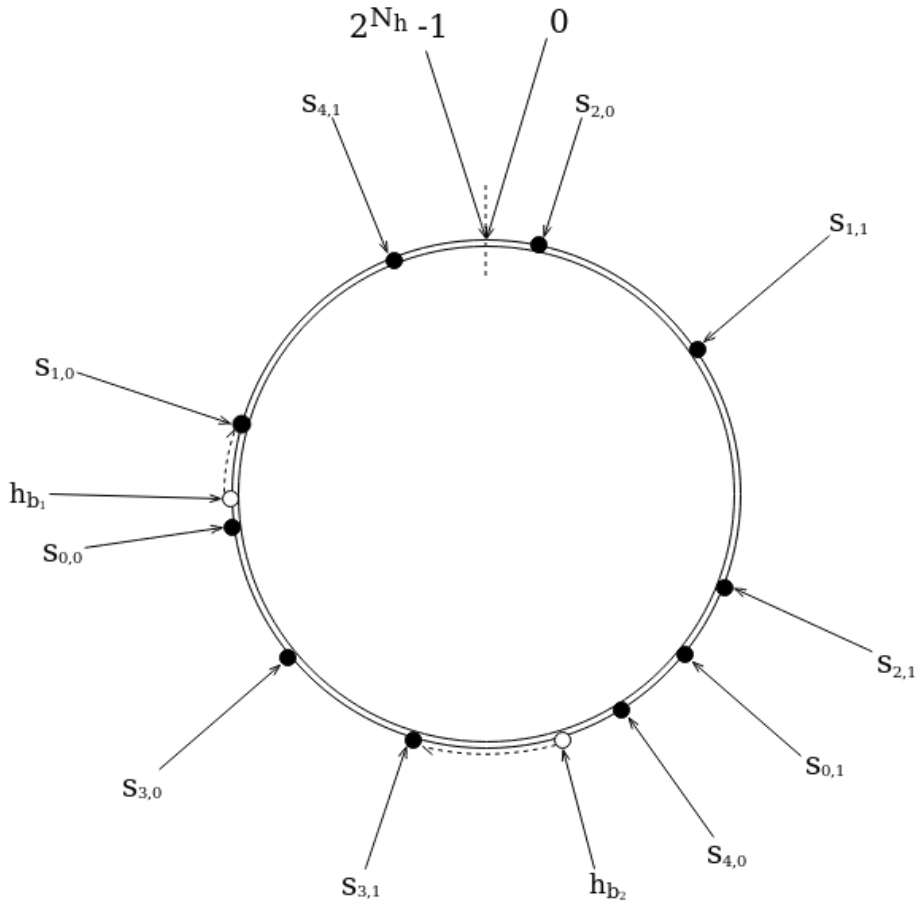


Figure 3.17: *The sharding procedure when virtual nodes are present, shown in a similar way as in Figure 3.12. In order to find which Agent is responsible for a given key, we have to scan the ring in the clockwise direction until a hash digest of a virtual node of some Agent is reached.*

Scaling the system out or in, works in the same way when virtual nodes are present, too. Again, the only difference is that when the Agent joins or leaves the cluster, there are multiple segments of the consistent hashing ring – exactly V segments: one for each one of the V virtual nodes of every Agent – that are being created from or merged with the rest of the ring segments. As far as the movement of stored data is concerned, the properties explained in [21] still hold, including the aforementioned *smoothness* and *monotonicity* properties. The major difference is that when virtual nodes are used, each ring node does not have a single adjacent ring node to pull the data from, as in Figure 3.13. It can rather have as many as $max\{V, N_s\}$ adjacent ring nodes from which it may have to pull data from when it joins the cluster.

Pictorially, in Figure 3.16, assuming that the fifth Agent ($i = 4$) has just joined the cluster, it needs to pull the data that lie in its assigned segments from both the third Agent ($i = 2$) since $S_{4,1}$ previously was a part of $S_{2,0}$, and the fourth Agent ($i = 3$) because $S_{4,0}$ was previously a part of $S_{3,1}$.

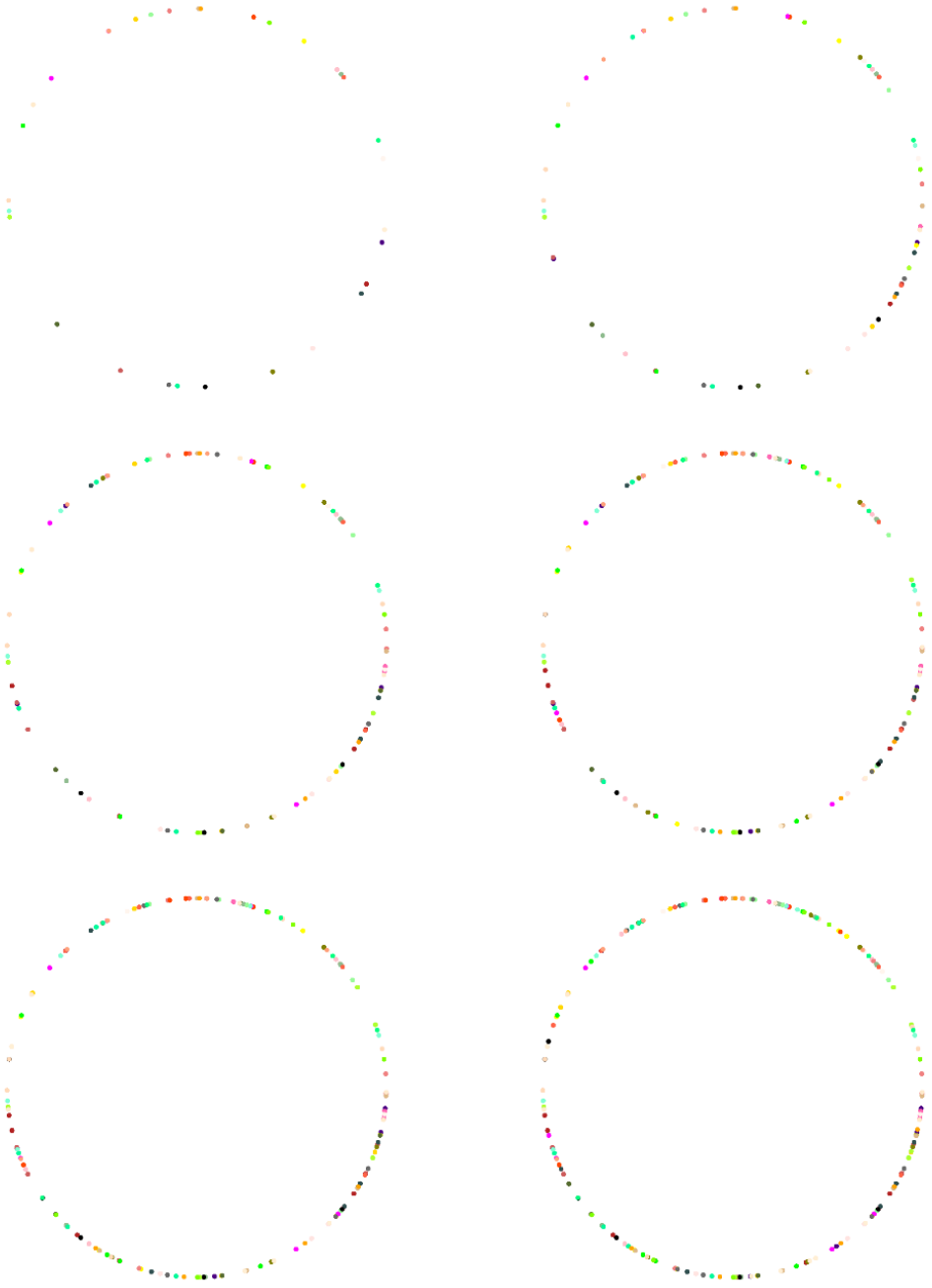


Figure 3.18: *Visualization attempt of a consistent hashing ring where $N_s = 32$ and the values of V are integers in the range $[1, 6]$ (left to right, top to bottom).*

Next, let us examine the load balancing in terms of "bins and balls" that has been men-

tioned so far for both MICAS' and RICAS' load balancing. As always, a ball corresponds to an object stored in the system. The presence of virtual nodes is reflected only indirectly in the model: a bin still corresponds to a single Agent, which is assumed to consist of multiple "virtual bins". We run another series of Monte Carlo simulations, where a constant value, $N_s = 1000$, is used for the number of bins, and various values are used for both V, the number of "virtual bins" per bin, and n, the number of balls. In Figure 3.19, Figure 3.20, Figure 3.21, Figure 3.22, Figure 3.23 and Figure 3.24, we can see how the number of balls per bin is distributed among the $N_s = 1000$ bins at the end of the simulations. It is evident that increasing the value of V while keeping n the same, results in more even distribution of the balls among the bins, confirming our intuition from the descriptions so far. Nevertheless, comparing Figure 3.23 and Figure 3.24 with Figure 3.2 and Figure 3.3, the corresponding diagrams for mod-N hashing, we can see that it is hard for RICAS to achieve the same efficiency in load balancing as MICAS' mod-N hashing, even when V is set to large values.

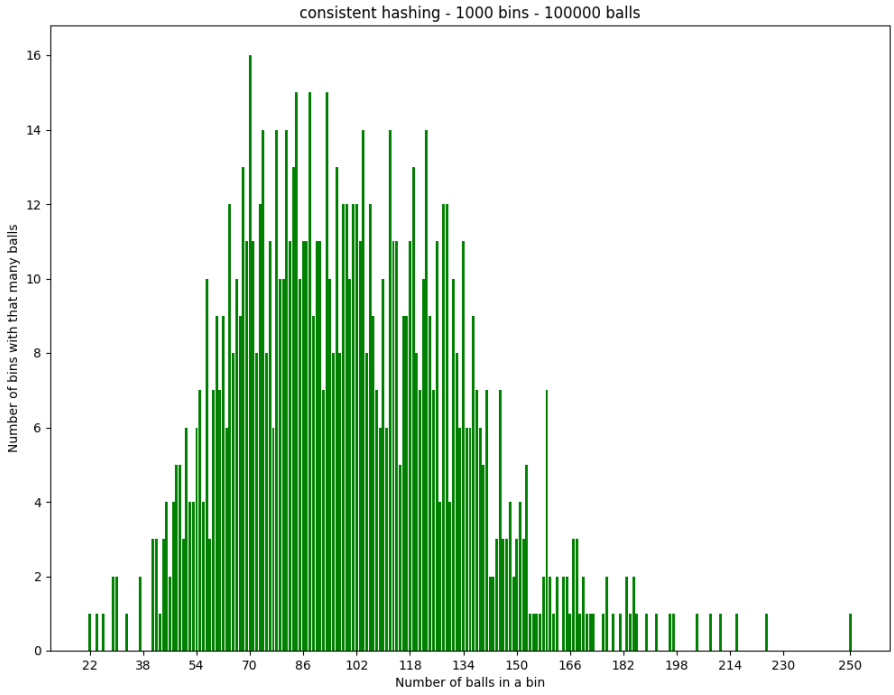


Figure 3.19: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins, $V = 10$ virtual bins and $n = 10^5$ balls. $\frac{n}{N_s} = 100$ balls per bin.*

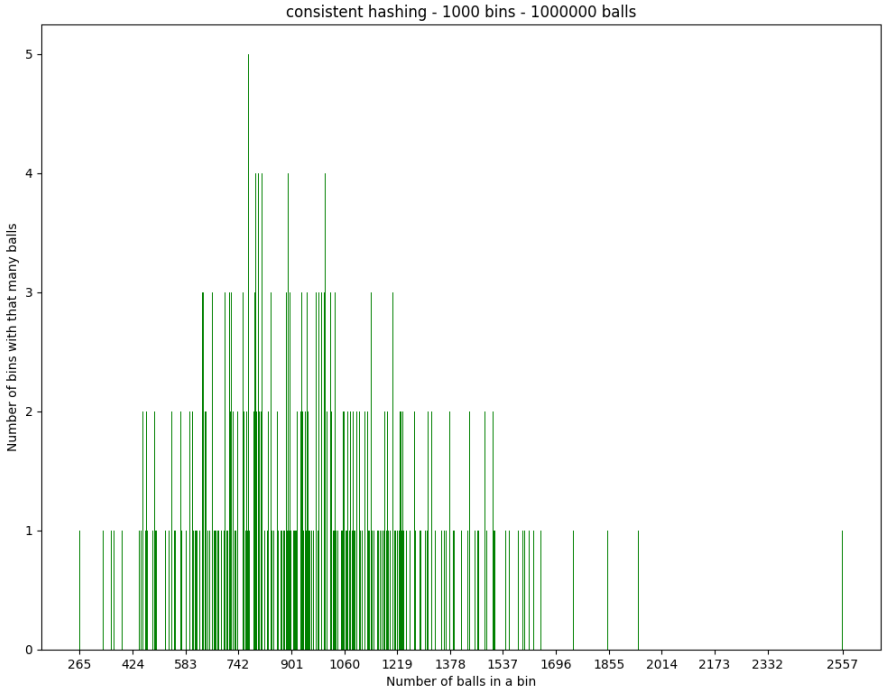


Figure 3.20: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins, $V = 10$ virtual bins and $n = 10^6$ balls. $\frac{n}{N_s} = 1000$ balls per bin.*

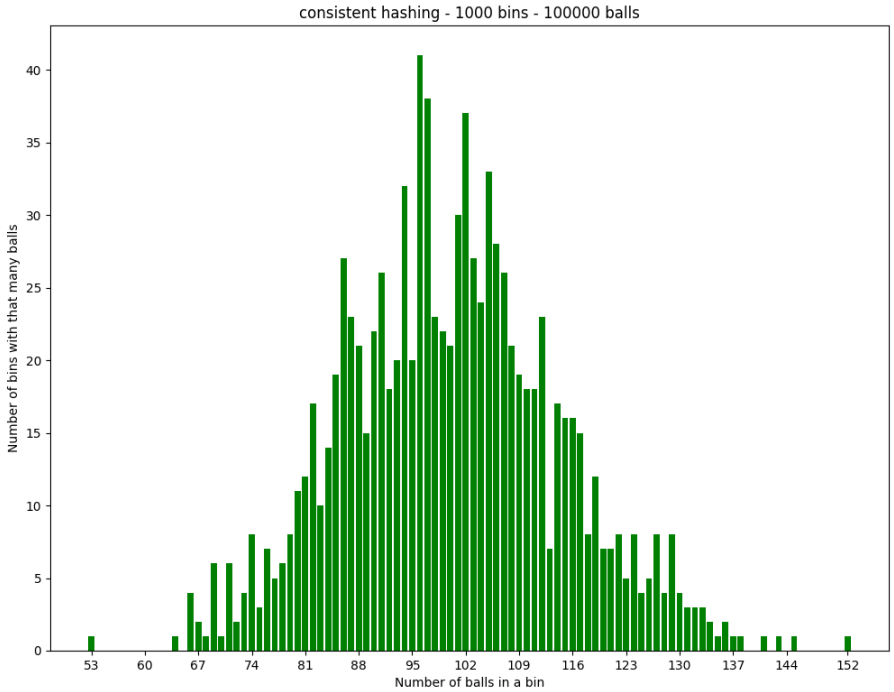


Figure 3.21: *Balls per bin distribution using* ***consistent hashing****, for* $N_s = 1000$ *bins,* $V = 100$ *virtual bins and* $n = 10^5$ *balls.* $\frac{n}{N_s} = 100$ *balls per bin.*

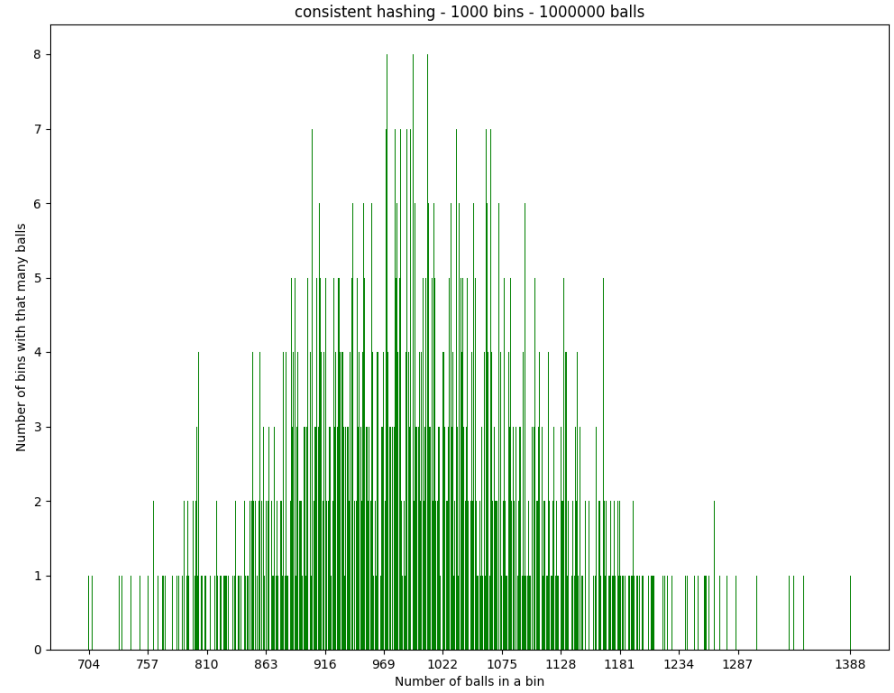


Figure 3.22: *Balls per bin distribution using* ***consistent hashing****, for* $N_s = 1000$ *bins,* $V = 100$ *virtual bins and* $n = 10^6$ *balls.* $\frac{n}{N_s} = 1000$ *balls per bin.*

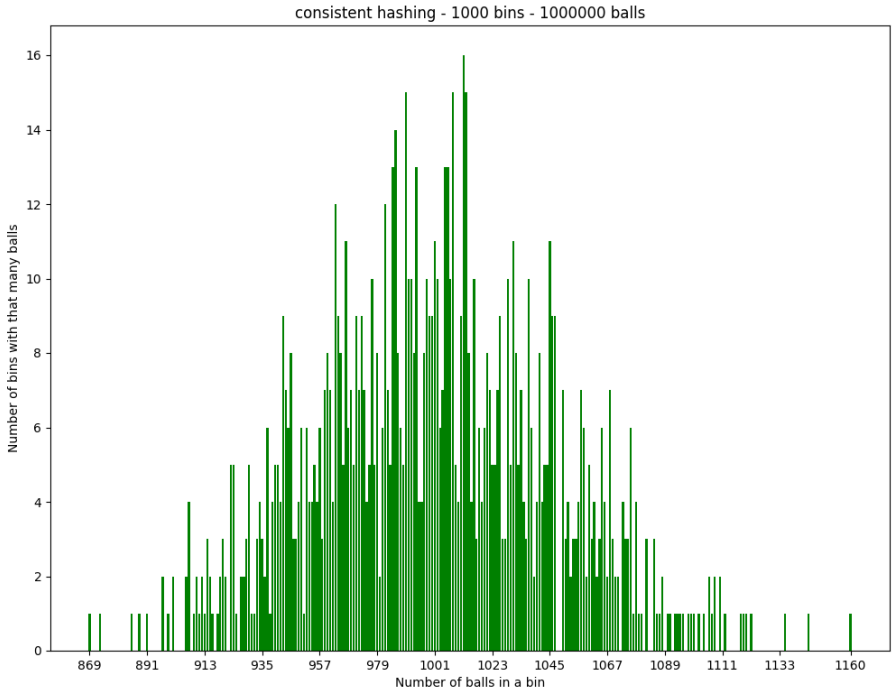


Figure 3.23: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins, $V = 1000$ virtual bins and $n = 10^6$ balls. $\frac{n}{N_s} = 1000$ balls per bin.*

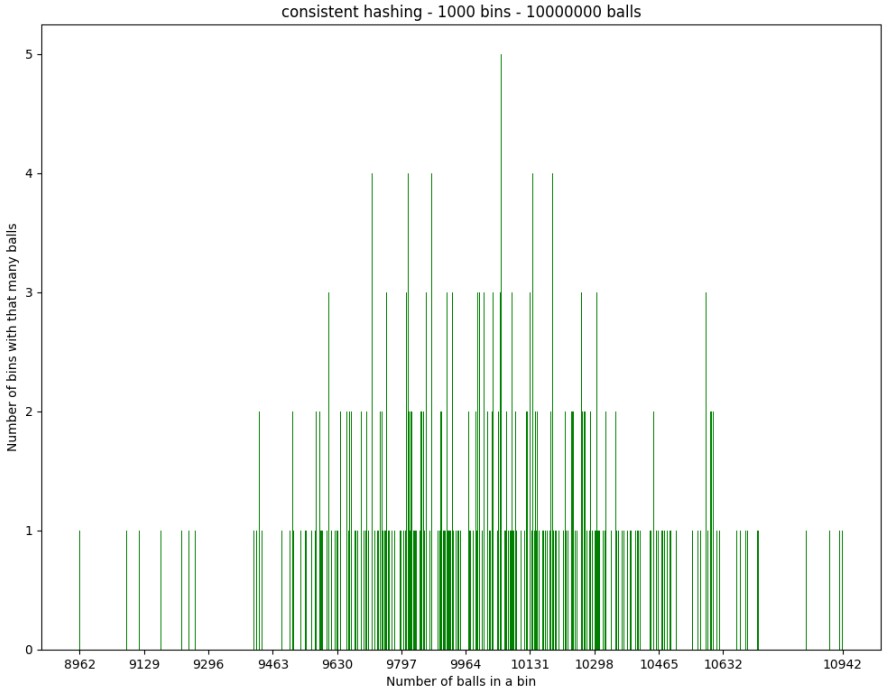


Figure 3.24: *Balls per bin distribution using **consistent hashing**, for $N_s = 1000$ bins, $V = 1000$ virtual bins and $n = 10^7$ balls. $\frac{n}{N_s} = 10000$ balls per bin.*

A reasonable question arises as to how the value of V, the number of virtual nodes per ring node, is selected. It is obvious that the presence of virtual nodes is beneficial for both the load balancing and the horizontal scalability of the system: the more they are, the better it is for the system. Is there a cost for using a lot of them, though?

In fact, there is, and it is related to some memory requirements and the computational complexity of some local operations on behalf of both the Agents and the Proxies in the cluster. As explained in section 3.7, every Agent needs to know the state of the cluster regarding the shards and the other Agents that are responsible for them. So do the Proxies, which traditionally need to forward the incoming `CREATE` and `READ` requests from the clients to the appropriate instances of the backend tier: every client request needs to consult this state so as to be correctly redirected to one – or more – Agents. Therefore, this consistent hashing data must be maintained in some sort of data structure so as to be often and easily accessible by all instances of the frontend and the backend tier of `RICAS`; otherwise, the expense of additional network hops to route a request to its correct destination might be incurred. As the number of virtual nodes per ring node is incremented, the size of the data structure is incremented as well. It is further increased considering that the data structure must also store information required for the replication process, which will be discussed soon, in subsection 3.9.2. Its size in memory may well reach the order of MBs. Over these, multiple $\mathcal{O}(\log(V \cdot N_s))$[28] search operations are performed routinely, most of which result in cache misses for the processors, especially considering the high competition for the cache that should regularly occur due to the nature of the system.

Now that the trade-offs related to the use of virtual nodes are understood, we are capable of appreciating the need of configuring V to a relatively small rather than a very large value. Fortunately, Karger et al. in [21] have already proven that it suffices V to be logarithmic to the number of shards in the system. Therefore,

$$V \approx \log_2 N_s \tag{3.6}$$

should usually be a good choice.

At this point, we have to mention two additional restrictions imposed by our imple-

[28] More on the data structure in section 4.4.

mentation of RICAS. Knowing about them should help readers have a clearer picture of the functionality of the system early on.

First, in the current version of RICAS, the number of virtual nodes, V must be configured statically at the time of the deployment of the system, and it cannot change later. At the same time, according to (3.6), the value of V should loosely depend on the value of N_s, which, however, is variable. In fact, the whole point of using consistent hashing is to allow for N_s to be variable and thus take advantage of the elasticity it can provide. In practice, the suggested value for V in RICAS is

$$V \approx \log_2 N_{s_{max}} \tag{3.7}$$

where $N_{s_{max}}$ is the maximum value that N_s is expected to reach throughout the use of the system.

At first, this limitation may seem similar to that of MICAS. In the case of MICAS, N_s is statically defined, thus it is required by the system administrator to know the maximum expected value of N_s. However, if N_s is set to a lower value than that at the time of the deployment, it is plain impossible to further scale MICAS out – it is impossible without having to rehash and relocate the entire data stored, anyway. In the case of RICAS, though, (3.7) is far less strict – it is more of a suggestion. Even when it does not hold true, the system still is infinitely scalable and elastic, anyway. It will merely perform less than ideally in terms of either load balancing of the data stored among the Agents (when set to a lower value), or efficiency of search operations performed by the Proxies for every client request (when set to a higher value).

As a matter of a fact, when $N_{s_{max}}$ is very low, the value of V provided by (3.7) might be too small. As we examined earlier, as the virtual nodes per ring node get fewer, the effectiveness of the load balancing is worsened. When both the number of ring nodes and the number of virtual nodes per ring node are small, the total number of virtual nodes on the ring is small, too, thereby affecting the load balancing significantly. In such cases that $N_{s_{max}}$ is considerably low, (3.7) should probably be ignored since the value of V should be appreciably higher than what it suggests.

The second restriction is that, in the current version of RICAS, the number of virtual nodes cannot be different between the Agents. In other words, every Agent, once

added to the cluster, it must be mapped exactly V times on the consistent hashing ring. As it was explained earlier, it is normally expected that every virtual node is assigned a $\frac{1}{V \cdot N_s}$ fraction of the objects stored. If it had been allowed to configure the number of virtual nodes at a per-Agent granularity, it would be theoretically possible to finely designate them according to each Agent's underlying storage capacity, yielding a

$$\frac{V_i}{\sum_{j=0}^{N_s-1} V_j}$$

fraction of the total objects for the i-th Agent that has V_i virtual nodes, where $i \in [0,\ N_s - 1] \subset \mathbb{Z}$. By further digging into this direction, though, we can see that there are new challenges being introduced, and since this thesis does not mean to focus on that, we move past it for now. Thereby, for the time being, we stick by our simplifying assumption that all physical nodes in the cluster have almost equal underlying storage capacity, as stated in section 3.6.

3.9.2 Replication

Replication is a vital feature of `RICAS` that aims to provide high availability of the stored data, by ensuring a certain degree of fault tolerance.

In `MICAS`, we saw "*full shard replication*", where each shard is replicated k times as a whole, k being the replication factor. Since every deployed Agent is responsible for one single shard in its entirety, full shard replication can be achieved merely by deploying k Agents per shard instead of one, and instructing the Proxies to reach all k of them when serving client requests for an object. However, the fact that each shard is further splitted into multiple smaller pieces in `RICAS`, due to the presence of virtual nodes, complicates the procedure of replication.

The usual approach to confront this peculiarity is actually quite different than the one in the case of `MICAS`, yet fairly simple. During an object's creation, first, the key is assigned to a virtual node as usual: by scanning the ring in a clockwise direction starting from the object's hash digest until a virtual node is met. To incorporate the replication technique of `RICAS`, the clockwise scan of the ring only stops when it reaches as many virtual nodes as needed, as long as these belong to k distinct ring nodes. A replica of

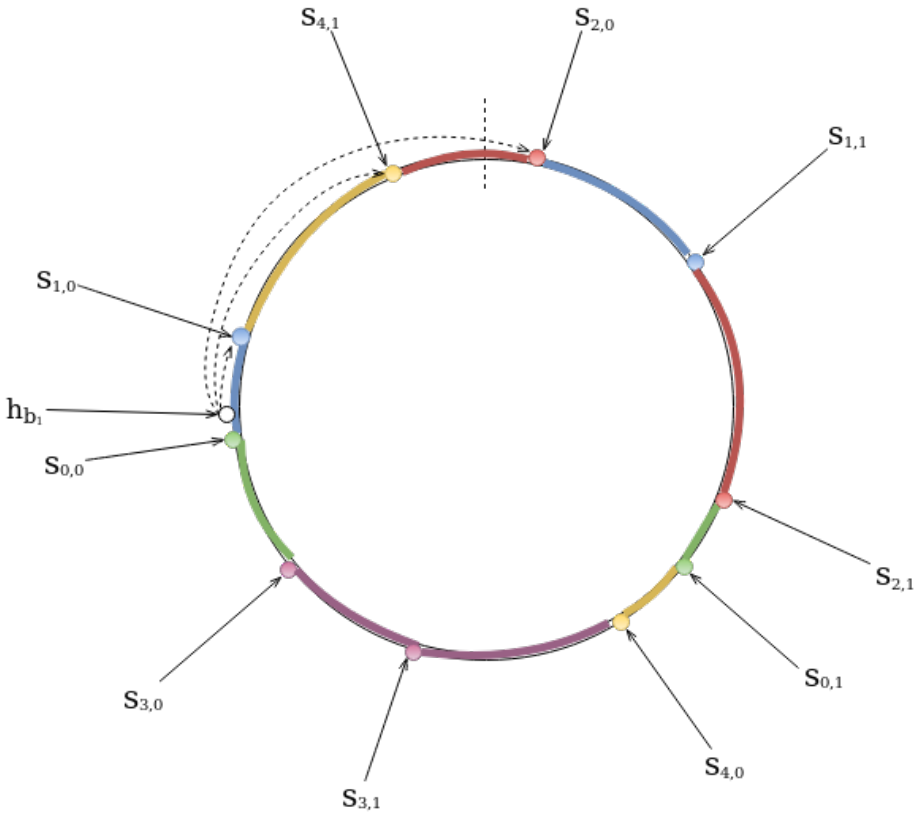


Figure 3.25: *A consistent hashing ring of $N_s = 5$ nodes and $V = 2$ virtual nodes each, with replication factor set to $k = 3$. The object b_1 is replicated to Agents S_1, S_4 and S_2 because the first three virtual nodes that are reached by scanning the ring in the clockwise direction from its key h_{b_1} are $S_{1,0}$, $S_{4,1}$ and $S_{2,0}$.*

the object is created in each one of these k distinct ring nodes. An example can be seen in Figure 3.25, where the replication factor is assumed to be $k = 3$ and blob b_1 hashes to $h_{b_1} = h(b_1)$ on the ring. By scanning it in a rightwards direction, first $S_{1,0}$ is met, which belongs to ring node S_1, then $S_{4,1}$ of ring node S_4, and last $S_{2,0}$ of ring node S_2. A replica of the object with key h_{b_1} is thereby assigned to all S_1, S_4 and S_2.

It is emphasized that the scan stops when k **distinct** ring nodes are met; not just k virtual nodes on the ring. There is a chance that more than one virtual nodes within a scan of the first k virtual nodes belong to the same ring node. In this case, some virtual nodes must be skipped and the scan must be continued, otherwise $k - 1$ fault tolerance may not be achieved.

To illustrate this, let us consider the example shown in Figure 3.26. The replication factor is still assumed to be $k = 3$, and another blob, b_2, that hashes to $h_{b_2} = h(b_2)$

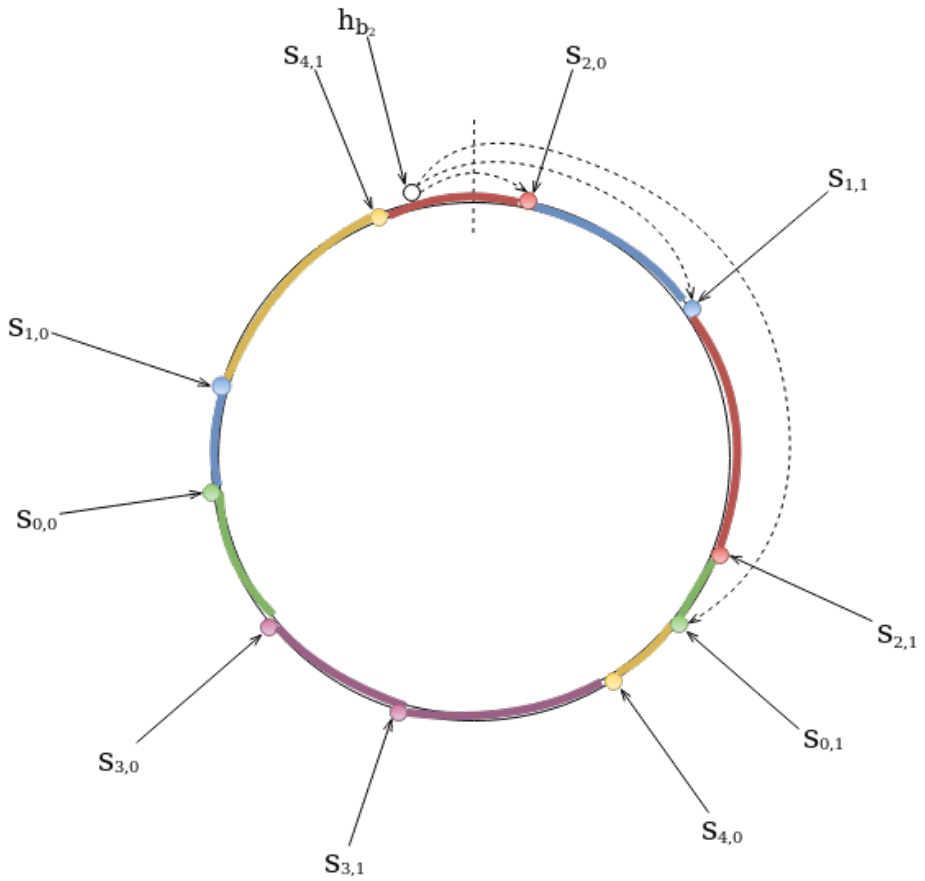


Figure 3.26: *The same consistent hashing ring as in Figure 3.25, but this time a virtual node has to be skipped until virtual nodes of $k = 3$ **distinct** ring nodes are reached. One can see that blob b_2 ends up being stored in S_2, S_1 and S_0 after reaching four (rather than three) virtual nodes by scanning the ring.*

is placed on the ring. Scanning the ring in a clockwise direction, virtual node $S_{2,0}$ is reached first, then $S_{1,1}$ and then $S_{2,1}$. Obviously, the first k virtual nodes that were reached during the scan correspond to only $k - 1$ distinct ring nodes: S_2 and S_1. Therefore, the system would be resilient to only $k - 2$ ring node failures regarding object h_{b_2}. To cope with that, as it was described above, the scan moves past virtual node $S_{2,1}$ until a k-th distinct ring node is reached. Indeed, virtual node $S_{0,1}$ of ring node S_0 is right next, so finally the object is assigned to k distinct ring nodes: S_2, S_1 and S_0, and the system is resilient to $k - 1$ failures.

This sort of replication technique, in conjuction with the presence of virtual nodes in the ring, also affects the initial data synchronization procedure of `RICAS`. However, these implications are presented later, in subsection 3.9.4, after `RICAS`' architecture is discussed.

3.9.3 Architecture

As explained in section 3.7, RICAS consists of two tiers: the frontend and the backend, as MICAS does. In contrast to MICAS, RICAS' architecture includes one more component, the **Operator**, which, although it is crucial for the functionality of the system, it is never on the data path, by design. More about the functionality of the Operator will be unravelled gradually in both this and the next subsection.

To begin with, the role of the Proxies is known from section 3.7. They are deployed as Kubernetes Pods controlled by a Deployement. As explained in section 3.7 and subsection 3.8.3, the Proxy Pods can be deployed on Kubernetes using either a Deployment or a DaemonSet, but using the former is our preferred method due to the greater ease it provides at scaling the tier out and in. Their functionality is pretty much similar to that in MICAS, with the exception that this time the sharding takes place using consistent hashing. Therefore, they should be aware of the state of the consistent hashing ring each time, in order to forward incoming requests from the clients towards the appropriate RICAS Agents. They should also be able to detect Agents joining or leaving the cluster, and update their local in-memory consistent hashing ring data structure accordingly.

The backend tier of RICAS is quite different than that of MICAS. While MICAS has a static number of shards, each of which is mapped to a separate StatefulSet of k MICAS Agents, where k is the replication factor, the number of shards in RICAS changes dynamically. To this end, every RICAS Agent is deployed in a Kubernetes Pod, and all of them are controlled by a single StatefulSet. The Replicas field under the Spec field of the StatefulSet API object denotes the current number of Agents in the cluster, N_s; hence, the number of distinct ring nodes in the consistent hashing ring. As usual, Kubernetes is responsible for assigning a unique ID to each particular Pod, an integer in the range $[0,\ N_s - 1]$. Based on this, the Agent Pods are reachable either via the Kubernetes Endpoints of the Headless Service associated with the StatefulSet, or using the DNS addresses that are created by KubeDNS.

At this point, we should discuss about the naming of the consistent hashing ring nodes and virtual nodes, a subject that had been postponed earlier in subsection 3.9.1. The obvious way to name an Agent, i.e. a ring node, is after the unique integer that is as-

signed to its corresponding Pod by Kubernetes, due to it being in the backend `StatefulSet`. Thereby, the unique name of the ring nodes is of the form "`Agent-i`", where $i \in [0,\ N_s - 1]$. Next, as we mentioned in subsection 3.9.1, we need to deterministically generate a sequence of V unique IDs for the V virtual nodes of every ring node, so that they can later be hashed and mapped onto the consistent hashing ring. This is achievable simply by appending another integer to the Agent's name, effectively enumerating all its virtual nodes. Therefore, the unique name of every virtual node in the system ends up being in the form "`Agent-i-j`", where $i \in [0,\ N_s - 1] \subset \mathbb{Z}$ and $j \in [0,\ V - 1] \subset \mathbb{Z}$.

The exact way that the Agent Pods are scheduled across the physical nodes that comprise the Kubernetes cluster is actually very important. As presented in subsection 3.9.2, the replication of the stored objects across the `RICAS` Agents depends entirely on the specific order that all virtual nodes, from all Agents, are laid out on the consistent hashing ring. It takes place in a kind of "random" way that also ensures decent balancing of the objects stored among the ring nodes. To retain resilience to $k-1$ node failures, i.e. in order for all objects stored in `RICAS` to be available to the clients even when $k - 1$ nodes are down, we have to make sure that no two replicas of the same object, in any virtual nodes across all Agents in the cluster, are actually located on the same underlying physical node. However, due to the presence of virtual nodes in the formation of the consistent hashing ring, and especially as their number gets higher, for every ring node, the union of all sets of virtual nodes that are adjacent[29] to its own virtual nodes, probably includes virtual nodes from every single ring node in the consistent hashing ring.

Moreover, the ability of the system to dynamically change the number of shards, N_s, raises additional concerns related to the scheduling of the Agents on physical nodes. As `RICAS` is scaled out or in and Agents are added to or removed from the cluster, additional virtual nodes are placed on or removed from the ring in a "random" manner (without any real pattern with respect to their layout on the ring), thus continually changing the adjacency relations among them. Namely, even if the N_s Agents are scheduled in such a way that a number of them are located on the same physical node

[29] "Adjacency" here is used as defined in subsection 3.9.2, where, during the process of sharding and replication for an object, the scanning of the consistent hashing ring for virtual nodes was not over until virtual nodes from k **distinct** ring nodes were met. These distinct ring nodes were finally assigned to be the replica owners of that object.

while no two replicas of an object are colocated, scaling the system in by, let us say, two Agents, might create new adjacency relations between Agents that were previously not logically adjacent on the ring, and thereby physically colocated. The objects that were previously assigned to the Agents that are being removed from the cluster, are now "densely" relocated to the fewer Agents that stay. Based on the currently applied replication technique, such a scale-in would bring some replicas of the same objects together on the same physical node, effectively decreasing the fault tolerance of the deployed `RICAS` system. In addition, after the procedure of the scale-in finishes, even if there was an easy way to schedule the Agents correctly in order to ensure the desired $k - 1$ fault tolerance having a number of them running on the same physical node, the cost of rescheduling and data movement might cancel out any benefits, especially if it happens often.

To completely overcome this kind of difficulties, the logic of scheduling is kept as simple as possible. We force Kubernetes scheduler to schedule every Agent Pod in the backend `StatefulSet` on a different physical node, strictly. To achieve that, we properly configure the relevant Kubernetes Pod affinity settings; in particular `Inter-Pod Anti-Affinity`[180], as we did in `MICAS` for a similar purpose, in conjunction with the appropriate Kubernetes `Labels`.

Scale-out and scale-in are two of the most important operations in a `RICAS` cluster, and largely what differentiates it from `MICAS` in terms of cluster manipulation operations. Notwithstanding their significance, they are operations that affect the stored data only indirectly, contrary to clients' `Create` requests. When a scale-out or scale-in operation is issued by an administrator, what is directly affected is actually the *metadata* of the cluster, which in turn results in the movement of stored data among the Agents. In the case of scale-out and scale-in, the relevant metadata of the cluster is the value of N_s, the number of shards. Such operations, that only affect the stored data indirectly, are henceforth considered operations of the **control plane**, or **control operations**. Contrary to control operations, operations like `Read`, `Create` and `List`, which act directly on the stored data, are considered operations of the **data plane**, or **data operations**.

Data operations are issued by clients of the system, explicitly. On the contrary, control operations may either be issued explicitly by the cluster administrator, or they may be

issued implicitly by the system itself, in the form of *cluster phases* transitions. Cluster phases were introduced in section 3.7, and will be thoroughly discussed later, in subsection 3.9.4. An example of such a control operation that takes place implicitly in `RICAS` is **garbage collection**, which will also be discussed later, in the next subsection.

There is at least one good reason to discuss about the distinction between control operations and data operations. Under our current data model, where Update operations on the data are not supported at all, operations on the control plane are the only ones that actually require **consensus** among the deployed `RICAS` Agents. Indeed, when scaling the system out or in, the movement of the data among the Agents can never be successfully completed unless all Agents are aware of the scaling operation that takes place. The Agents become aware of that essentially by "keeping an eye" on the cluster metadata for changes (using the `watch` API of the Kubernetes API server) – for now we can simplistically assume this is just N_s in the case of scaling operations – and acting on them in whatever way they deem necessary.

The coordination between the Agents takes place in the form of "cluster phases" and "Agent phases". Their mechanics will be examined in detail in subsection 3.9.4. Until then, we have to consider that this kind of state information has to be persistently stored somewhere, so that any `RICAS` Agent that fails and then gets restarted, knows where it left off and is able to adjust its behavior accordingly. The way this state information is updated is another crucial characteristic: every Agent needs to be aware of all other Agents' current state – or at least the most recent – in order to function properly and to update its own state whenever it is required. Therefore, any modifications on the shared state information have to be ordered, and they have to be immediately visible to all Agents in this order.

Fortunately, Kubernetes can cover all these needs. As it has been mentioned before, in subsection 3.8.4, Kubernetes provides *optimistic concurrency control* through its API objects and their *resource versions*, taking advantage of the linearizability provided by its backing store, etcd. Using Kubernetes' `CustomResourceDefinitions`, also known as `CRD`s, a new Kubernetes API resource is defined, dedicated to store `RICAS` cluster's and Agents' state information and metadata. Hereinafter, we are going to refer to this custom API object as `RicasCluster`. In a similar fashion to all Kubernetes API objects, `RicasCluster` is accessible via standard Kubernetes' API: its definition

includes, among others, the `Spec` and `Status` fields, which, as usually, reflect the desired and the current state of the API object, respectively.

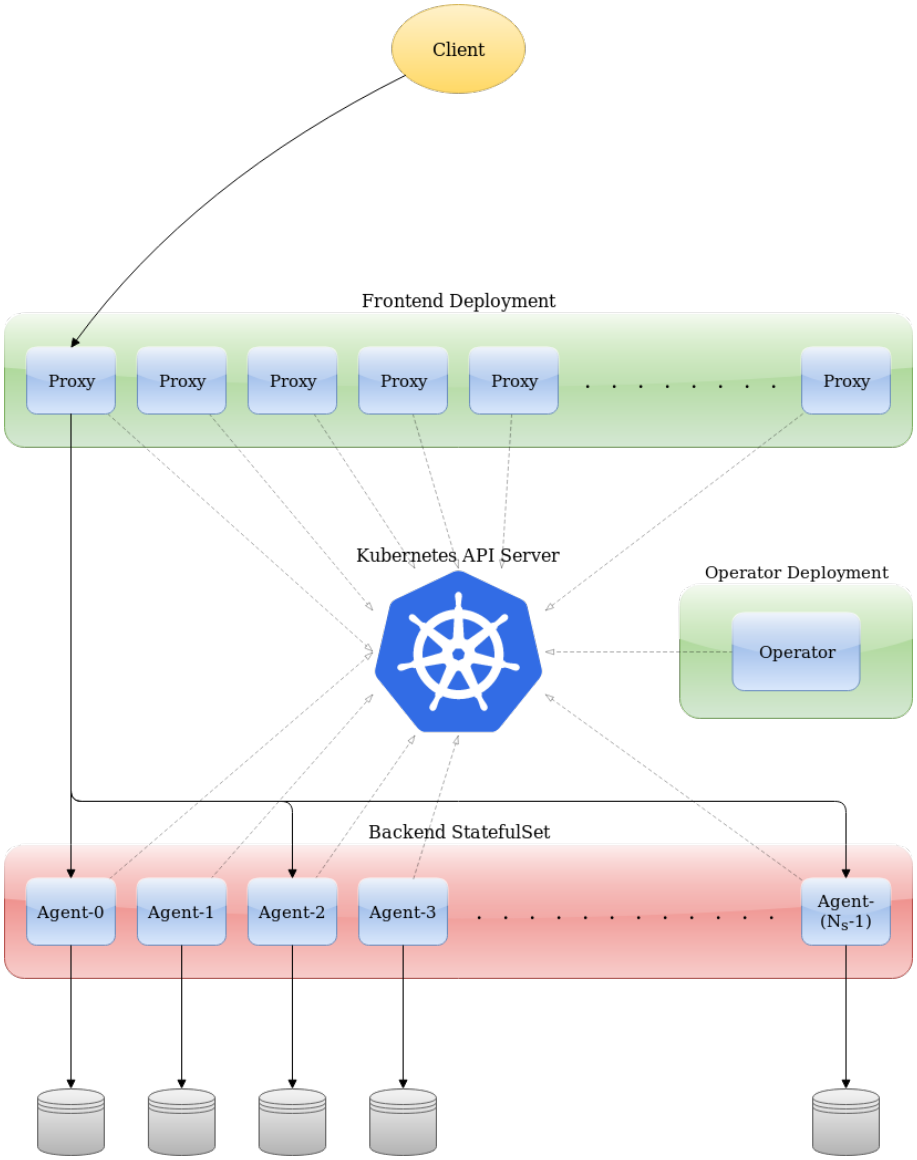


Figure 3.27: *The proposed architecture of* `RICAS`*, for a deployment consisting of an arbitrary number of Proxies, N_s shards, and the Operator.*

`RicasCluster` is supposed to be the standard way for the cluster administrator to interact with a deployed `RICAS` cluster. All current and future control operations have to be initiated through this. In the same spirit as all Kubernetes API objects, the interaction is declarative in nature: the cluster administrator declares the desired state in its `Spec` field, and `RICAS`, with the help of Kubernetes, makes sure that the current state is properly reconciled. This is where the role of the Operator fits in as well. Simply put, the Operator makes sure that all interactions make sense and are legit according

to the phases that the cluster and all its Agents are in. All this information is still a little vague, but it will hopefully be made clear by the end of this chapter.

Figure 3.27 illustrates the architecture of RICAS. There is an arbitrary number of Proxy Pods controlled by the "Frontend Deployment" and N_s Agent Pods controlled by the "Backend StatefulSet", which is only one in the case of RICAS. There is also a separate Deployment that controls a single Pod for the Operator. The dashed arrows indicate communication on the control plane, and they appear to connect every RICAS component to the Kubernetes API server – this is where the RicasCluster object can be accessed from. The rest of the arrows indicate an example of communication on the data plane, i.e. the flow of stored, or to-be-stored, data (during a Read or a Create request, respectively), where the replication factor is assumed to be $k = 3$.

3.9.4 Cluster phases, Failover & Recovery

In the same spirit as MICAS, RICAS has to be highly available, hence fault tolerant. Since RICAS has more features than MICAS, as it has been presented so far, achieving resilience to failures throughout the extra functionality is a more complex procedure that requires extra cautiousness.

The state diagram for a RICAS cluster is illustrated in Figure 3.28. The diagram and its phases should reveal to the reader the "big picture" about the functionality of the cluster at any point in time. However, the phase that the RICAS cluster is in and the phases that all of its Agents are in, are actually interdependent. Therefore, we should dive into the logic of the Agents, and observe what their own phases are, and what states do they represent.

The state diagram for a RICAS Agent is shown in Figure 3.29. Evidently, the number of phases that a RICAS Agent may be in are more than those of a MICAS Agent, which were discussed in subsection 3.8.4. All of them are going to be explained, along with the logic of transitioning from one to another and the actions that are required to be taken during these transitions.

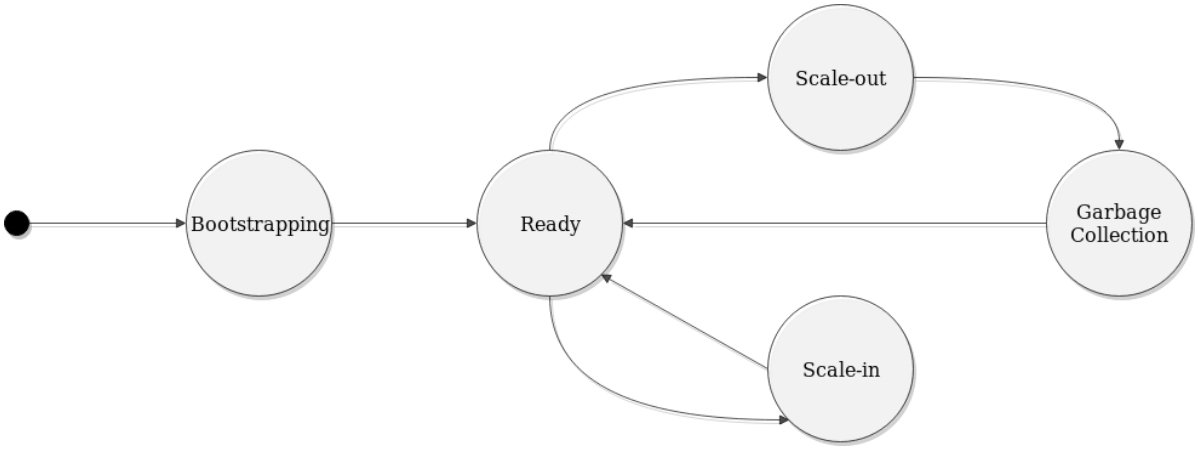


Figure 3.28: *All possible phases and phase transitions of a* `RICAS` *cluster deployment. Their design and implementation are related to the* `RicasCluster` *custom Kubernetes API resource.*

3.9.4.1 Bootstrapping & Ready phases

Let us begin with the bootstrapping of the cluster, which is similar, to a certain extent, to that of `MICAS`. When the `RICAS` cluster is first booted, i.e. all Agent, Proxy and Operator Pods have just been deployed and they are about to run for the first time, all Agents enter the **Ready phase** as they come up. However, the cluster as a whole is said to be in the **Bootstrapping phase**. The cluster is supposed to transition from the Bootstrapping phase to its own **Ready phase** only when all of its component entities are up and ready to serve any incoming requests.

To achieve this transition, we use a `Bootstrapping` flag, as we did in the case of `MICAS`, although this time the procedure is a little different. The `Bootstrapping` flag is stored in the `.Status.Bootstrapping` field of our custom `RicasCluster` API object, since it is actually a part of the metadata of the `RICAS` cluster. Every time an Agent boots, even in cases of a failover, it checks whether the `Bootstrapping` flag of its associated `RicasCluster` is raised, to deduce whether the cluster is in the Bootstrapping phase or not; its actions at this point actually depend on the phase that the cluster is in, as we will examine soon. When all Agents are finally up for the first time, this phase is over for the cluster, and the corresponding flag must be cleared. This is a job for the Operator: when the `RICAS` cluster is first deployed, the Operator starts watching for changes in the state of the backend `StatefulSet`. Specifically, when the value of `.Status.ReadyReplicas` of the backend `StatefulSet` becomes equal to the value of its `.Spec.Replicas` field (which in turn is equal to N_s and also tracked

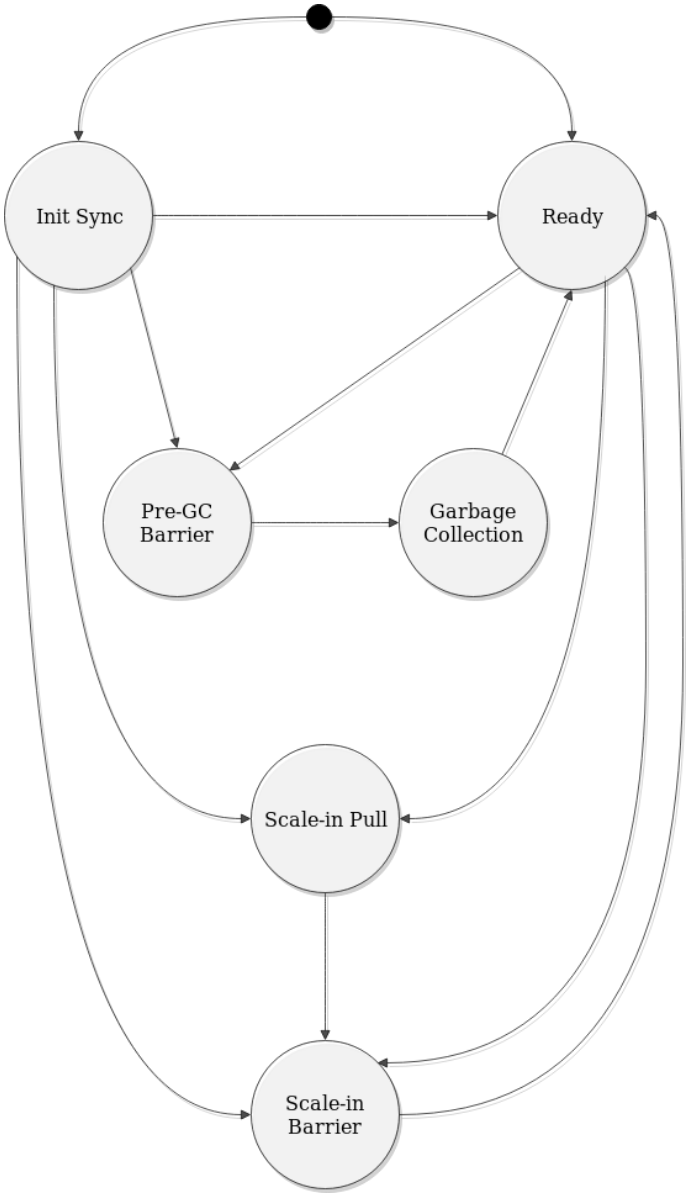


Figure 3.29: *All possible phases and phase transitions of a deployed* `RICAS` *Agent.*

in a `.Spec.RingSize` field of the `RicasCluster` object), the Operator is responsible for clearing the Bootstrapping flag [30].

The Ready phase of the cluster, as shown in Figure 3.28, indicates simply that the cluster is operating normally, and (almost) nothing "interesting" is occurring. That is to say, there are no pending scale-in, scale-out or garbage collection operations to be

[30] In fact, to completely avoid the minor race condition that occurs when the Operator becomes aware of the end of the Bootstrapping phase before the last Agent deployed does, the Operator waits some time before it clears the flag from the `RicasCluster` API object. This time frame may henceforth be referred to as `bootGracePeriod`, and is configurable at compile-time.

completed. On the other hand, the Ready phase of an Agent, as shown in Figure 3.29, indicates that that Agent is operating normally, and nothing "interesting" is happening from its own point of view. This means that all the Agent is doing is simply serving client requests as they arrive from the Proxies. It is important to point out that in order for the cluster to be in the Ready phase, it is not required that all of its Agents are in the Ready phase too. Agents may fail for any reason, at any time, and they are restarted by Kubernetes. During the time that a number of Agents are down, hence not Ready, it is possible that no scaling or garbage collection operation takes place, therefore the cluster might be in the Ready phase. Yet, when these Agents are restarted, they are expected to complete their initial data synchronization to recover any data they might be missing, and then proceed to the Ready phase again.

3.9.4.2 Initial Data Synchronization & Init Sync phase

Having set the reader straight about the distinction above, let us examine the other phase that an Agent might find itself in upon booting. This is the **Init Sync phase**, a phase that has multiple aspects in different parts of the functionality of `RICAS`.

In the simplest case, an Agent enters it when it starts running, after it observes that the Bootstrapping flag is not raised in its associated `RicasCluster` object, hence the cluster is not in the Bootstrapping phase. From the aspect that we are currently examining it, the phase is essentially similar to what was referred to as *Syncing phase* in the context of `MICAS`: it is the state in which the Agent performs its initial data synchronization. Using **rsync** for efficiency (in a way that will be discussed in more detail in section 4.5), the Agent downloads any replicas of objects that it is responsible for and does not already have stored (probably because they were created during its absence), from other Agents that are up and Ready, and also have such replicas according to the replication algorithm presented in subsection 3.9.2. When the initial data synchronization is over, the Agent transitions to the Ready phase.

The process of initial data synchronization of the `RICAS` Agents is pretty similar to that of `MICAS`; it has been previously mentioned in subsection 3.7.2 and explained in subsection 3.8.4 for `MICAS`. Every `RICAS` Agent Pod consists of two Docker containers, as shown in Figure 3.30. One of them exists to contain a process that implements the logic for our own `RICAS` Agent, and the other one to contain an always-listening rsync

server running in daemon mode. The job of the latter is simple: to export a single rsync module that exposes the root directory where the Agent stores all the data that it is responsible for. These data can then be served to any rsync client connection directed to that module.

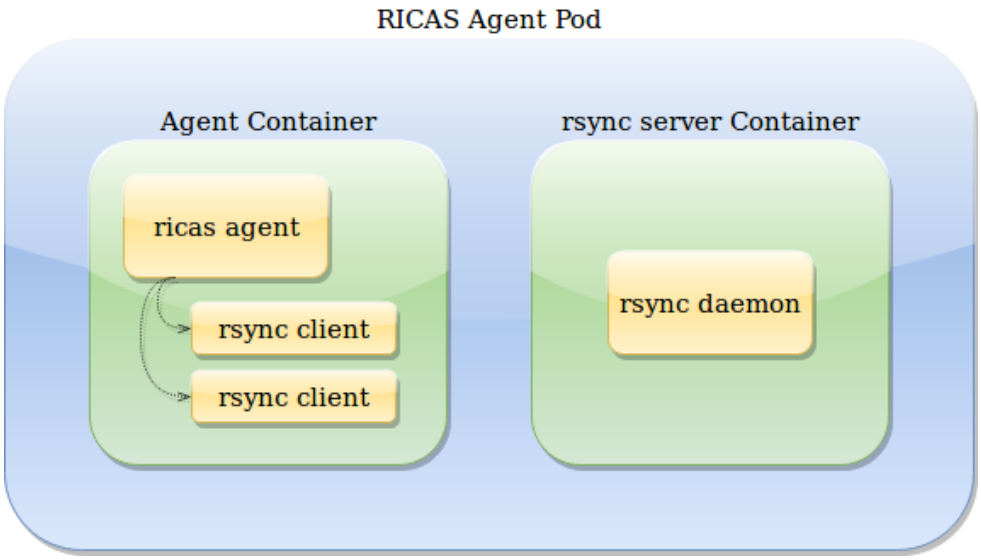


Figure 3.30: RICAS *Agent Pod.*

A freshly spawned Agent entering the Init Sync phase, first retrieves the current state of the RICAS cluster, by retrieving the `RicasCluster` API object from the Kubernetes API server. Based on that information, it can fully reconstruct the current state of the consistent hashing ring data structure in its own memory, knowing where all virtual nodes are placed on it and what is the value of the replication factor. Next, it traverses the ring in a clockwise direction, examining whether itself is responsible for storing replicas of the objects in each one of the virtual nodes that it comes across. For every such virtual node, it forks off an rsync client process to connect to the rsync daemon of one of the other replica owners that is up and Ready, in order to download the corresponding replicas. When the whole consistent hashing ring is traversed and all rsync client processes have finished, the Agent has completed its initial data synchronization and transitions to the Ready phase.

Before we dive into the rest of the cluster and Agent states, we have to go through the scale-out and scale-in operations, and describe how they actually take place in RICAS. To do that, first we need to explictly define what has been so far vaguely referred to as *cluster state* or *cluster metadata*.

3.9.4.3 RicasCluster: Our Custom Kubernetes API Resource

An important part of the design of RICAS, is to extract an amount of information that minimally, yet accurately, represents the state of the cluster.

As it has been stated before, this information is stored by Kubernetes in the form of a custom API resource, which is available through the standard Kubernetes API and created by the means of a `Custom Resource Definition`. This custom resource in the case of RICAS is the aforementioned `RicasCluster`. In the same spirit as all Kubernetes API resources, `RicasCluster` has a `Spec` and a `Status` field, which describe the desired and the current state of a `RicasCluster` API object, respectively. The definition of the `RicasCluster` custom API resource in Go is shown in the Listing below.

```
type RicasCluster struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   RicasClusterSpec   `json:"spec"`
        Status RicasClusterStatus `json:"status"`
}
```

Listing 3.1: *Definition of the* `RicasCluster` *custom API Resource in Go.*

At this time, we are interested in the design of RICAS regarding its state information; hence, diving into the gory details of a Kubernetes API object would be off-topic. Instead, we mean to draw the reader's attention to specific fields that help us to make our point, and ignore others that do not contribute to it. It is obvious from the above Listing that the `Spec` and `Status` fields of the `RicasCluster` resource are custom types, which are specific to RICAS.

```
type RicasClusterSpec struct {
        RingSize          int32 `json:"ringSize"`
        ReplicationFactor int32 `json:"replicationFactor"`
        VirtualNodeCount  int32 `json:"virtualNodeCount"`

        IndexHeight       int32  `json:"indexHeight,omitempty"`
```

```
        InternalTimeoutMs int32  `json:"internalTimeout,omitempty"`
        SyncerBackend     string `json:"syncerBackend,omitempty"`
        SyncerWorkers     int32  `json:"syncerWorkers,omitempty"`
        StorageBackend    string `json:"storageBackend,omitempty"`
        RsyncIOTimeout    int32  `json:"rsyncIOTimeout,omitempty"`
        RsyncConnTimeout  int32  `json:"rsyncConnTimeout,omitempty"`

        ProxyCount int32 `json:"proxyCount"`
}
```

Listing 3.2: *Definition of* `RicasClusterSpec` *type, which corresponds to the* `Spec` *field of a* `RicasCluster` *custom Kubernetes API resource, in Go.*

In Listing 3.2 we present the definition of the `RicasClusterSpec` type, which corresponds to the `Spec` field of every `RicasCluster` custom API resource. Most of the fields concern the configuration of a `RICAS` deployment at boot time; and their functionality may or may not be obvious yet. We have to emphasize on two of them: the `RingSize` and `ProxyCount` fields. The `RingSize` field is an integer that indicates the value of N_s, the desired number of Agents (hence shards in the system) that should exist in the deployed `RICAS` cluster. The `ProxyCount` field, likewise, is another integer that indicates the desired number of Proxies that should be up. These two fields are configurable by the cluster administrator while the system is in use, to scale the backend or the frontend of the `RICAS` deployement. Their value can even be edited via the standard `kubectl edit` command, which is provided in most typical Kubernetes installations.

```
type RicasClusterStatus struct {
        RingSize          int32 `json:"ringSize,omitempty"`
        ProxyCount        int32 `json:"proxyCount,omitempty"`
        Bootstrapping     bool  `json:"bootstrapping,omitempty"`
        GarbageCollecting bool  `json:"garbageCollecting,omitempty"`
        ScaleOutFactor    int32 `json:"scaleOutFactor,omitempty"`
        ScaleInFactor     int32 `json:"scaleInFactor,omitempty"`
        ScaleInUnfinished int32 `json:"scaleInUnfinished,omitempty"`

```

```
10          SyncStatus                *BitSet `json:"syncStatus"`
11          ScaleOutRole              *BitSet `json:"scaleOutRole"`
12          ScaleInPending            *BitSet `json:"scaleInPending"`
13          GarbageCollectionPending *BitSet
                `json:"pendingGarbageCollection"`
14  }
```

Listing 3.3: *Definition of* `RicasClusterStatus` *type, which corresponds to the* `Status` *field of a* `RicasCluster` *custom Kubernetes API resource, in Go.*

In Listing 3.3 we present the definition of the `RicasClusterStatus` type, which corresponds to the `Status` field of a `RicasCluster` API object. These fields constitute the current **state of the** `RICAS` **cluster** as perceived by its components: the Agents, the Proxies and the Operator. All of them need to have access to this state, but only the Agents and the Operator may modify it. None of these fields should ever be modified by an administrator; their values are supposed to be meaningful only to the system itself (even though some of these values may actually appear meaningful to the administrator, too).

Since almost all of them are important for the scaling and the garbage collection operations of `RICAS` (in fact, all but `ProxyCount`), a brief description for each one of them is provided right below:

- `RingSize`
  An integer value which, together with `ScaleOutFactor` and `ScaleInFactor`, represent the value of N_s, the current number of deployed `RICAS` Agents, at any time. As we will see soon, its value may be modified only by the Operator in response to a change in the `.Spec.RingSize` field of the `RicasCluster`, in order to initiate a scale-out or a scale-in control operation.

- `ProxyCount`
  An integer value that indicates the current number of `RICAS` Proxies that are up and serving client requests. Its value is supposed to be modified only by the Operator in response to a change in the `.Spec.ProxyCount` field of the `RicasCluster` to scale the frontend tier out or in, by deploying additional or removing existing Proxy Pods, respectively.

- `Bootstrapping`

  This is the Bootstrapping flag, which was presented earlier in this subsection. It is a boolean value that is actually useful only during – and for a short time after – the `RICAS` cluster has been deployed for the first time, to indicate whether the cluster has transitioned from the Bootstrapping to the Ready phase or not. It is modified only by the Operator.

- `GarbageCollecting`

  This is the garbage collection flag, a boolean value that indicates whether the cluster is in the Garbage Collection phase or not. It may be modified only by the `RICAS` Agents. Its usage will be made clear in 3.9.4.4 and 3.9.4.5.

- `ScaleOutFactor`

  An integer value – although not a "factor" in the mathematical sense – that is nonzero only when the `RICAS` cluster is in the process of scaling out. When it is nonzero, its value indicates the number of Agents by which the system is being scaled out, i.e. the number of Agents that are being added to the cluster. When its value is zero, either there is no scale-out operation occurring in the cluster, or there is one which ends soon (the bits in the `SyncStatus` make the difference here, as we will examine later). It is modified by both the Operator and the Agents, but it is also useful to the Proxies, as we are going to discuss in 3.9.4.4 and 3.9.4.7.

- `ScaleInFactor`

  An integer value – also not a "factor" in the mathematical sense – that is nonzero only while the `RICAS` cluster is in the process of scaling in. During that time, its value indicates the number of Agents by which the system is being scaled in, i.e. the number of Agents that are being removed from the cluster. When its value is zero, one cannot assume that there is no scale-in operation occuring in the cluster unless the value of the `ScaleInUnfinished` field is zero as well. It is modified by both the Operator and the Agents, but it is also useful to the Proxies, as we will see soon, in 3.9.4.6 and 3.9.4.7.

- `ScaleInUnfinished`

  An integer value, although used mostly as a boolean, that indicates whether the

most recent scale-in operation on the RICAS cluster is really finished or not. It is modified by both the Operator and the Agents. Its existence is made necessary by the fact that the scale-in operation is a little more complex than the scale-out in that the Agents must not be removed from the cluster before all of their data have been successfully transferred to other ones. We will see more about it in 3.9.4.6.

- SyncStatus

  A bit array, where the i-th entry indicates whether the i-th Agent is in the process of synchronizing its data with the data of its peers or not. Every time an Agent's data replicas are not completely up to date with the replicas in the rest of the replicas in the cluster, that Agent's corresponding bit in this bit array signifies that the Agent is unreliable to be used as a source in other Agents' data synchronization procedure. This includes the aforementioned case of the Init Sync phase. SyncStatus may be modified by both the Agents (the i-th Agent modifies only the i-th entry) and the Operator (it can modify all of the entries). More about it will be discussed in 3.9.4.4 and 3.9.4.6.

- ScaleOutRole

  A bit array, where the i-th entry, during the process of a scale-out operation, indicates whether the i-th Agent has completed retrieving all the data that it is responsible for according to the new state of the consistent hashing ring, at least once since the scale-out operation began. It may be modified by both the Agents (the i-th Agent modifies only the i-th entry) and the Operator (it can modify all of the entries). Its functionality will be presented in 3.9.4.4.

- ScaleInPending

  A bit array, where the i-th entry, during the process of a scale-in operation, indicates whether the i-th Agent has completed retrieving all the data that will be responsible for thereafter, according to the new state of the consistent hashing ring, or not. It may be modified by both the Agents (the i-th Agent modifies only the i-th entry) and the Operator (it can modify all of the entries). We will see more of its usage in 3.9.4.6.

- GarbageCollectionPending

  A bit array, where the i-th entry indicates whether the i-th Agent is ready to tran-

sition to the Garbage Collection phase. It may be modified only by the Agents (the i-th Agent modifies only the i-th entry). Its functionality will be made clear in 3.9.4.4 and 3.9.4.5.

Another critical point that has already been explained, but must be emphasized, is how the state is actually modified.

All this cluster state data are persisted as a custom Kubernetes resource, and they are handled in the same way as the standard Kubernetes RESTful API, over HTTP. The Kubernetes API server provides *optimistic concurrency control* for its API objects, using *resource versions*. All `PUT` operations on our `RicasCluster` object carry a specific value that represents the version of that object over which the modification is attempted. If, while handling an incoming `PUT` request, the API server realizes that the resource version of the associated stored object is different than the one found in the request, it deduces that some other `PUT` operation has been successfully applied first, and thereby this `PUT` operation is rejected and its sender is notified with an HTTP error `409 CONFLICT`. As a result, all the operations on the `RicasCluster` object are ordered, and all the components that issued them are always aware of the latest state of the `RicasCluster` before they take their action onto it.

This is a principal element in the design of the state and its manipulation in `RICAS`, upon which multiple aspects of the design of the whole system are based. It is the element that enables the successful coordination among the Agents, and with the Operator, so that no data are lost, even in dangerous cases of multiple Agent or node failures. Based on this behavior, and using the `watch` API that is provided by the Kubernetes API server in both the Agents and the Operator (implemented as *Controllers*[31].), it is guaranteed that the transitions between the phases shown in Figure 3.28 and Figure 3.29 are always accurate and timely.

3.9.4.4 Scale-Out

Adhering to the principles of Kubernetes, the scaling of a `RICAS` cluster is issued in a declarative way.

[31]The *controller pattern*, based on the concept of the *reconciliation loop*, has been discussed earlier, in subsubsection 2.2.3.2

The administrator is expected to simply modify the value of the field `.Spec.RingSize` of the `RicasCluster` object associated with the `RICAS` deployment in question, for instance using the `kubectl edit` command, replacing the current number of deployed Agents, N_s, with the desired number, $N'_s > N_s$, thus effectively asking for $M = N'_s - N_s$ extra Agents to be deployed. It has to be noted that the Operator does not allow such a change to occur while another scale-out, scale-in or garbage collection operation is in place. The Operator is aware of the exact state of the cluster – with respect to these operations – by watching the `Status` field of the `RicasCluster` object, and appropriately rolls back the state of the `RicasCluster` in case of any illegitimate scale-out attempt.

When the Operator notices this change and verifies that the timing is acceptable, itself is the `RICAS` component that initiates the scale-out operation. In a single `PUT` request to the Kubernetes API server, it sets the value of the field `.Status.ScaleOutFactor` of the `RicasCluster` object to M, and flips the bits in the range $[N_s,\ N'_s - 1]$[32] of the `.Status.SyncStatus` and `.Status.ScaleOutRole` bit arrays [33]. The `RICAS` cluster transitions from the Ready phase to the **Scale-out phase**, as per Figure 3.28. Furthermore, the Operator properly modifies the `.Spec.Replicas` field of the backend `StatefulSet` to notify Kubernetes that it should scale it up by deploying M additional Agent Pods.

In the context of the scale-out operation – and only as long as this takes place – all Agents are conceptually divided into two groups. The **Junior Agents**, or simply **Juniors**, are the M Agents that are being added to the cluster as a result of the current scale-out operation. The **Elder Agents**, or simply **Elders**, are the N_s Agents that already existed in the cluster before the current scale-out operation was issued. This is the exact meaning of the entries in the `.Status.ScaleOutRole` bit array: its i-th entry indicates whether the i-th Agent acts as a Junior or an Elder.

Thanks to the monotonicity property, when a `RICAS` cluster is scaled out, data may only move from the Elders towards the Juniors. When the Juniors join the cluster, they have to "claim" the data they are responsible for in the new state of the ring from

[32] Indexing is assumed to be starting from zero.

[33] Although the initial size of the `SyncStatus` and `ScaleOutRole` bit arrays, as well as the size of all the other bit arrays in the `RicasCluster`, is configurable only at compile time for now, in the future this can be easily implemented to either be configurable at the time of the deployment of the system, or to automatically grow and shrink according to its needs.

the Elders that had them stored previously. This data transfer procedure is really just a slight variation of the initial data synchronization as we have examined it so far; hence, the Junior enters the **Init Sync phase**. The only difference with the recovery procedure of a restarted Agent, as far as the transfer of the data is concerned, is that, this time, a Junior also needs to know the k Agents – where k is the replication factor – which previously were the replica owners of its newly assigned objects. Therefore, instead of consulting only the current state of the consistent hashing ring, i.e. its state including the M newly added Agents, it also needs to take into account the ring's state as it was before the scale-out operation was issued, when it included only the N_s Elder Agents.

The transfer of the data does not have any particular interest. It takes place using rsync, in the same way as it was described earlier in this subsection, in 3.9.4.2. Now that the reader is aware of the existence of the `.Status.SyncStatus` field, we emphasize that the transfer of the data is considered reliable, and thus finished, only when the Agent that the object replicas are pulled from, is up to date regarding its stored data.

Whenever a Junior finishes its initial data synchronization during a scale-out, it flips again its own bits in the `.Status.SyncStatus` and `.Status.ScaleOutRole` bit arrays of the `RicasCluster`, to signify that from now on it may act as an Elder once the Scale-out phase is over, and that its data are, thereafter, up to date (until the Agent fails or has a pending scale-in operation to handle, as we will see later). At the same time, it checks whether itself is the last remaining Junior Agent of this scale-out procedure, i.e. that all the other Agents that were added to the cluster as a result of the same scale-out operation have already finished their initial data synchronization. If this is the case, it means that the scale-out operation is successfully coming to an end, so this last Junior also sets the value of the `.Status.ScaleOutFactor` to zero, in the same `PUT` operation as its `.Status.ScaleOutRole` bit flip. However, it should be noted that the Scale-out phase is considered finished only when all these hold true:

- The `ScaleOutFactor` has been zeroed.
- All M Juniors' `ScaleOutRole` bits indicate that they can be considered Elders.
- All M Juniors' `SyncStatus` bits indicate that they are not currently in the middle of an initial data synchronization pull.

At first glance, the procedure may appear overly complicated for no apparent reason. But this is untrue. To realize that, one has to consider the various scenarios of failure that may occur at any time and any component of the system during the scale-out procedure.

First, let us examine the potential failure scenarios for the Juniors. Trivially, outside the context of the scale-out operation, there is no "Junior" Agent at all, and all Agents are equals; we only care about failures that occur during the scale-out process. When the Junior enters the Init Sync phase, the logic of the initial data synchronization, i.e. the logic that defines which Elders should be queried for each portion of the data, depends on the Agent's ID (which, in our case, is the integer in the range $[0,\ N_s - 1]$ assigned by Kubernetes to all Pods in the backend `StatefulSet`) rather than its bit in the `ScaleOutRole` bit array. Therefore, whether this bit has been flipped or not at the time of the failure, is irrelevant to its handling. When a Junior Agent fails, it simply restarts (actually, it is restarted by Kubernetes, as we already know), enters the Init Sync phase, as usual, checks whether the `RicasCluster` is in the Scale-out phase, and begins the initial data synchronization properly. What the `ScaleOutRole` really indicates, is whether every Junior has completed its initial data synchronization at least once since the beginning of the scale-out procedure.

Put simply, a Junior may fail either *before* or *after* it flips its `ScaleOutRole` bit:

- In the former case, the `ScaleOutFactor` will never be zeroed until all Juniors' `ScaleOutRole` bits are flipped. Thereby, no special treatment is required: the effect of the failure is simply as if the restarted Junior Agent had been delayed at the first time it was booted.

- In the latter case, when the Junior restarts and begins synchronizing its data again, it is possible that all other Juniors complete their own initial data synchronization successfully before this Junior does. In this case, all `ScaleOutRole` bits indicate that all Agents are ready to act as Elders and the `ScaleOutFactor` is zeroed, too, even though the Scale-out phase should obviously not be considered to be over, since there is a Junior still pulling data. This is where the need for all Juniors – actually, all Agents, as we will see in 3.9.4.5 – to have their `SyncStatus` bit indicating that they are not in the process of retrieving data, in order

for the Scale-out phase to be considered finished and for the `RicasCluster` to transition to the next phase according to Figure 3.28.

Since the initial data synchronization of each Junior never depends on the status of other Juniors, by design, the system is resilient to as many as M Junior failures at a time. That is to say, all Juniors during a scale-out phase can be failed simultaneously, with neither causing inconsistency to the data stored by `RICAS`, nor affecting the number of replicas stored for each object, hence with affecting neither the overall availability of the stored objects, nor the fault tolerance of the N_s Elder Agents that are already present at the beginning of the scale-out operation.

Next, let us examine the potential failure scenarios for the Elders. Again, trivially, outside the context of the scale-out operation there are no "Elder" Agents at all; therefore, we are concerned only about failures that occur during the scale-out process. As we have already described above, thanks to the monotonicity property, the Elders do not have to retrieve any new data during a scale-out. It is them, however, that the Juniors retrieve their own newly assigned data from. Consequently, an untimely failure of an Elder Agent may possibly disrupt the initial data synchronization of one or more Juniors. However, assuming the replication is enabled, i.e. that the replication factor has been configured to be greater than one ($k > 1$) at the time of the deployment of the system so as to achieve $k - 1$ fault tolerance, the Junior has multiple ways to proceed. Even if one of the Elder Agents that it is pulling its data from fails, there must be $k - 1$ more Agents that own replicas of the same objects, for all objects. In other words, by `RICAS`' design, not only the whole scale-out procedure does not affect the resilience of the existing Agents in any way, but, in fact, it is this resilience that the Juniors rely on to carry out their initial data synchronization fast and correctly.

An Elder Agent that is up and running during a scale-out, is in Ready phase, according to Figure 3.29. Its rsync daemon serves any incoming data synchronization requests from the Juniors. When it fails, it transitions to the Init Sync phase, as usual, and properly sets its own `SyncStatus` bit in time, to indicate that it is not reliable for any data retrieval until it is done with its own initial data synchronization. This is important for the Juniors; we can be sure that no Junior retrieves incomplete data from a failed-over Elder whose object replicas are not up to date yet. Such an Elder's initial data synchronization, in turn, is carried out by contacting only other Elder Agents, so that

we can also be sure that the Elder does not retrieve incomplete data from a recently added Junior whose object replicas may not be up to date yet, either. When the initial data synchronization is over, the Elder transitions to the Ready phase, and flips its bit in the `SyncStatus` bit array. Note that until this happens, as we mentioned earlier, the `RicasCluster` is still considered to be in the Scale-out phase, even if all Juniors have completed their own data retrieval.
Next, it transitions to the Garbage Collection phase.

3.9.4.5 Garbage Collection

When a `RICAS` deployment is scaled out by adding Agents to the cluster, the existing data that are being stored are spread among the new number of ring nodes. In fact, as we examined in detail above, due to the aforementioned monotonicity property, the existing data may move only from the Elders to the Juniors, i.e. from the old Agents (those who were already in the cluster before the scale-out happened) towards the newly added Agents. Assuming that the system was in a consistent state when the scale-out operation was issued, and regardless of the actual value of the replication factor, k, there should always be a number of Agents ("Elder" Agents, as per 3.9.4.4), which, at the end of the scale-out, still store replicas of objects that do not need to store anymore, since the freshly joined Agents should be storing them instead from now on. These data on those Agents are no longer accessible by the Proxies because they are not among the k replicas of their respective object that are expected to be found by consulting the updated in-memory consistent hashing ring data structure. Therefore, they are useless, and keeping them around only burdens `RICAS` and the underlying nodes in terms of storage capacity. *Garbage collection* is the process of methodically getting rid of those excess data.

It should be obvious that garbage collection is only required at certain times. Under the normal operation of a `RICAS` deployment, when no scale-out or scale-in procedures are in place, every Agent is responsible for specific portions of the data, depending on how its virtual nodes are laid out on the ring. It is the addition of new virtual nodes to the ring what disturbs the arrangement of the object replicas, rendering some of the existing replicas superfluous. Therefore, the garbage collection should always take place every time that a scale-out operation is completed successfully. Indeed, as we

can see in Figure 3.28, a `RicasCluster` being in the Scale-out phase can only ever transition to the **Garbage Collection phase**.

From the point of view of the Agents, they all must agree on when the Garbage Collection phase should be initiated; in other words, consensus among them is required. If the garbage collection begins prematurely, it can potentially harm both the Juniors and the Elders. If a Junior has not completed its initial data synchronization and the rest of the Agents start garbage collecting data, the Junior may retrieve incomplete portions of the data from the Elders, if it relies on Elders that are not supposed to carry around the replicas in these portions of the data in the new state of the consistent hashing ring. On the other hand, an Elder may have failed and been restarted during the process of a scale-out. In case that its initial data synchronization starts before the scale-out is completed – thus is based on the old state of the consistent hashing ring – and the garbage collection starts before the initial data synchronization is over, the Elder may either perform an ineffective garbage collection due to race conditions between the two procedures, or worse, it may retrieve incomplete portions of its designated data without noticing, hence harming the fault tolerance of the system.

Perhaps the complexity of the scale-out operation regarding the state information in `RicasCluster` seems more reasonable now. During the last steps of the scale-out operation, whenever an Agent notices that the `ScaleOutFactor` has been zeroed (thus, also, all `ScaleOutRole` bits must have already been properly flipped), if itself is not in the middle of a data synchronization (regardless of whether it was an Elder or a Junior in the context of the prior scale-out operation) it sets its corresponding bit in the `GarbageCollectionPending` bit array of the associated `RicasCluster` API object. By doing so, it lets all other Agents know that it is ready to initiate the garbage collection locally for its own stored object replicas. When that happens, we say that the Agent transitions to the **Pre-GC Barrier phase**, according to Figure 3.29. If, during the scale-out, an Agent was a normally operating Elder, i.e. not one that failed, it must be transitioning from the Ready phase. If it was either a Junior, or an Elder that failed and restarted during the process of the scale-out, its prior phase must have been the Init Sync phase.

The Pre-GC Barrier phase is exactly what its name suggests: a distributed barrier among all deployed Agents that takes place right before the Garbage Collection begins.

The moment that the last Agent that had not set its GarbageCollectionPending bit yet is about to do so, by observing the status of the RicasCluster it should be able to deduce that:

- the bits in the ScaleOutRole indicate that all Agents, including itself, may act as Elders,
- the ScaleOutFactor is zeroed,
- the bits in the SyncStatus indicate that no Agent is in the middle of a data synchronization, including itself,
- the bits in the GarbageCollectionPending indicate that all Agents but itself, are ready to initiate the garbage collection locally.

Together with setting its own GarbageCollectionPending bit, this last Agent also raises the garbage collection flag in RicasCluster API object, by setting its relevant boolean field .Status.GarbageCollecting, transitioning the RicasCluster to the Garbage Collection phase.

When all Agents have reached the barrier and they observe that the RicasCluster is in the Garbage Collection phase, all Agents initiate the garbage collection locally. It is the time that all of them transition to the Garbage Collection phase, as shown in Figure 3.29. The actual implementation of the garbage collection depends on the exact way that the objects are stored, of course, i.e. on the storage engine that is being used. Abstractly, the garbage collector is an entity, e.g. a thread, in every Agent that traverses the consistent hashing ring in its latest state, removing all object replicas that are stored in the Agent that lie outside of its own designated ring segments, as these are formed by the arrangement of the virtual nodes on the ring.

Every time that an Agent finishes its local garbage collection, it clears its own bit in the GarbageCollectionPending bit array of the associated RicasCluster object, to let the other Agents know about it. This is the time that the Agent transitions back to the Ready phase. When the last Agent that had not finished its local garbage collection yet is about to clear its own bit in the GarbageCollectionPending bit array, it notices that all other Agents have already done so. Hence, it deduces that the RicasCluster should transition from the Garbage Collection phase back to the Ready phase,

as shown in Figure 3.28, so it also clears the associated `GarbageCollecting` field to signify that.

Let us consider the failure scenarios during the process of the garbage collection.

- If an Agent fails before its `GarbageCollectionPending` bit is set, it is basically a failure during the Scale-out phase before even the `ScaleOutFactor` was zeroed, which is described in 3.9.4.4.

- If an Agent is restarted from a failure and finds both its `GarbageCollectionPending` bit set and the `GarbageCollecting` flag in the `RicasCluster` raised, it deduces that it must have failed while it was in the middle of a local garbage collection procedure. Therefore, it restarts the local garbage collection, concurrently to the initial data synchronization and then goes on operating normally. Since these two processes touch object replicas that lie in different segments on the consistent hashing ring anyway, the outcome of their concurrent activation cannot lead to any inconsistency regarding the stored data. Although whether an Agent in this state should be considered to be in the Init Sync phase or in the Garbage Collection phase is indeed kind of indistinct, adding an extra phase in Figure 3.29 just to represent this case would make the diagram unecessarily more complex than needed, without major benefits to the reasoning about the situation.

- If an Agent is restarted from a failure and finds that its `GarbageCollectionPending` bit is set, but the `GarbageCollecting` flag of the `RicasCluster` is not raised, it deduces that when the failure occurred, itself had already acknowledged that the scale-out procedure is about to be completed and that it has been ready to initiate the garbage collection locally. While performing its data synchronization, it watches for the `GarbageCollecting` flag: when it is raised, it initiates the local garbage collection concurrently to the initial data synchronization. Similarly to the above case, these two procedures being active concurrently does not cause any inconsistency regarding the stored data, since each one of them deals with different virtual nodes' ring segments than the other.

3.9.4.6 Scale-In

In the same way as in the case of scale-out, the scale-in operations are also issued in a declarative way.

Again, the administrator modifies the value of the `.Spec.RingSize` field of the `RicasCluster` API object, e.g. using the `kubectl edit` command, replacing the current number of deployed Agents, N_s, with the desired number, $N'_s < N_s$, effectively requesting that $M = N_s - N'_s$ of the deployed Agents are terminated. It is noted, again, that the Operator does not allow such a modification to occur while another scale-out, scale-in or garbage collection operation is in place. The Operator is aware of the exact state of the cluster by watching the `Status` field of the `RicasCluster` object, and it rolls the `Spec` field back to its previous state in the case of any illegitimate scale-in attempt.

When the Operator notices this change and verifies that the timing is acceptable, it is the component that initiates the scale-in operation. In a single `PUT` request to the Kubernetes API server, it sets the value of the `.Status.ScaleInFactor` field of the `RicasCluster` object to M, and also flips the bits in the range $[0,\ N_s - 1]$ of the `SyncStatus` and `ScaleInPending` bit arrays. The `RICAS` cluster transitions from the Ready phase to the **Scale-in phase**, according to Figure 3.28.

In the context of the scale-in operation – and only as long as this takes place – all Agents are conceptually divided into two groups. The **Leaver Agents**, or simply **Leavers**, are the M Agents that are about to be removed from the cluster as a result of the current scale-in operation. The **Stayer Agents**, or simply **Stayers**, are the N'_s Agents that are going to remain in the cluster after the scale-in operation finishes.

So far, we have selected to found all sorts of data movement on a pull-based model based on rsync, in both `MICAS` and `RICAS`. To leverage the presence of rsync daemons, which are needed anyway for the initial data synchronization, we follow the same pull-based model in the case of scale-in operations, too. Due to the monotonicity property, which has been repeatedly explained, when a `RICAS` cluster is scaled in, data may only move from the Leavers towards the Stayers. Thus, in the Scale-in phase every Stayer must pull from the Leavers all the extra object replicas that it should own according to the new state of the consistent hashing ring, using rsync client processes to commu-

nicate with their rsync daemons. We have to note, however, that, as an optimization aiming to reduce the data movement from the nodes that host the Leavers, we take advantage of the fact that, sometimes, replicas of the same objects that a Stayer must acquire, may be also owned by other Stayers. In other words, in practice, during a scale-in, data are transferred towards the Stayers from either the Leavers or some other Stayers; but, of course, no data are ever transferred towards any Leaver.

Now we know the reason why the Operator does not modify the `StatefulSet` in the case of a scale-in, contrary to the case of scale-out. If it did, some of the Agents – the M Leavers – would be terminated right away. However, the Leavers must stay around for longer to allow Stayers to pull from them any new data that they should be storing thereafter. If the `ScaleInFactor` (i.e. the number of Agents that the `RICAS` cluster is being scaled in by) is lower than the replication factor, k, then replicas of all of the objects stored by the Leaver Agents should also be stored by some other Stayers. In this case, terminating the Leavers early would break the guarantee of $k - 1$ fault tolerance throughout the whole duration of the use of a `RICAS` cluster deployment. Put simply, if those "other Stayers" – the ones that store replicas of the same objects as the Leavers do – fail, for any reason, these objects will be totally unavailable until they come up again, harming both the high availability guarantee towards the clients of the system, and also delaying the completion of the pending scale-in process. If the `ScaleInFactor` is greater than or equal to the replication factor k, it is almost certain that there are some objects, replicas of which are stored only by Leaver Agents. In this case, if the Leavers were terminated early, not only would it be disastrous for the availability of the objects regarding the handling of client requests, but it would also render the scale-in operation impossible to ever be completed correctly, since there would be no way for some of the objects to be acquired by the Stayers at all.

Paraphrasing our previous description, the Operator makes sure that all Stayers have their `SyncStatus` bits indicating that they are in the process of synchronizing their stored data, and also their `ScaleInPending` bits indicating that they have not completed the scale-in pull of the current scale-in operation yet. When a normally operating (i.e. not a recently failed or restarted) Stayer Agent notices these changes, it transitions from the Ready phase to the **Scale-in Pull phase**, as per Figure 3.29. In this phase, the consistent hashing ring as it was in its old state (i.e. before the scale-in

operation was issued) is traversed and compared to the ring as it is in its new state (i.e. as it will be after the scale-in finishes). This way, the Stayer Agent can figure out exactly which data are thenceforth assigned to itself, hence which data should be pulled and from which Agents. When all these data are retrieved, the Agent switches back its own `ScaleInPending` bit in the `RicasCluster` API object to let every other Agent know of its status. It also flips its own `SyncStatus` bit to also indicate that it is not in the process of synchronizing its stored data anymore.

When that happens, the Stayer Agent transitions from the Scale-in Pull phase to the **Scale-in Barrier phase**. This is another "distributed barrier" among the deployed Agents, similar to the Pre-GC Barrier that was described earlier. All Stayers must complete their scale-in data pulls before the Leavers are terminated. This is what the Scale-in Barrier is useful for: every Stayer that finishes its own scale-in pull waits at the barrier for the other Stayers. When all of them reach this point, everyone knows that the Leavers are allowed to be terminated without causing any fault tolerance guarantee to be compromised. As a matter of a fact, since the Leaver Agents do not need to pull any data during the scale-in, they transition to the Scale-in Barrier phase directly from their prior phase when the scale-in operation is issued. Essentially, they cooperate with the Stayers by waiting for them for as long as required to finish their scale-in pulls.

When all Agents reach the Scale-in Barrier, there are two things that have to be done. First, the `ScaleInFactor` must be zeroed, and second, the backend `StatefulSet` must be scaled down from N_s to N'_s Agent Pods, to terminate the M Leaver Agents. These two actions refer to two distinct Kubernetes API objects; however, the scale-in operation cannot be considered finished until both of them are done. This is where the `ScaleInUnfinished` field in the `RicasCluster` object plays its part. It has to be noted that the `ScaleInFactor` must be zeroed at the same time that the last bit in the `ScaleInPending` bit array is flipped, hence by the last Stayer to flip its own bit, within the same single `PUT` request issued towards the Kubernetes API server. Otherwise race conditions would occur: there would be a possibility, albeit admittedly slight, that a Stayer Agent would fail and would be restarted "untimely", causing the whole Scale-in phase to be blocked forever, unfinished throughout the cluster. We will examine the behavior of the system during Agent failures – actually possible ones – a little later.

Let us examine this part of the procedure, the Scale-in Barrier phase, in greater detail. Once an Agent reaches this phase, it simply blocks, polling the `RicasCluster` data structure until it notices that all bits in the `ScaleInPending` bit array indicate that every Agent has reached the barrier and that the `ScaleInFactor` has been zeroed [34]. Thus, the polling begins when the scale-in pull is over for each of the Stayers, and immediately when the scale-in operation is issued for each of the Leavers. After the `ScaleInFactor` is zeroed, all Agents race to scale down the backend `StatefulSet` by setting the value of the deployed Agent Pod replicas to N'_s. When this is successfully over, all remaining Agents race to zero the value of the `ScaleInUnfinished` field of the `RicasCluster` object. Therefore, the `RICAS` cluster has truly exited the Scale-in phase, and transitions back to the Ready phase, as per Figure 3.28, only when `ScaleInUnfinished` is zeroed; a zero value in the `ScaleInFactor` field while the `ScaleInUnfinished` is non-zero indicates that the scale-in pull is over for all Stayer Agents, but the backend `StatefulSet` may not have been scaled down just yet. It is remarked that all those "races" above can be handled perfectly well, thus they may only yield well defined behavior, thanks to the optimistic concurrency control semantics provided by Kubernetes' API server.

The whole scale-in procedure may appear unreasonably complicated in terms of state manipulation actions at first. However, all these steps exist to ensure resilience to failures of any Agent, at any time during the procedure. Let us begin by examining what happens in the `RICAS` cluster when failures of Stayer Agents occur.

A Stayer can fail after the Operator has switched its `SyncStatus` and `ScaleInPending` bits in the `RicasCluster` object. Whether the Stayer has started its scale-in pull or not is not important; though, for now, we assume that the failure occurs while the Agent is still in the Scale-in Pull phase, before it reaches the Scale-in Barrier phase. When the Stayer Agent is restarted, it enters the Init Sync phase, as usual, to retrieve any data it may be missing, based on the state of the consistent hashing ring before the scale-in operation was issued. However, the Agent immediately notices its own `ScaleInPending` bit and also queues the appropriate scale-in pull to take place after

[34] In practice, the polling takes place in a much more efficient way than what it appears, through the use of Kubernetes' `client-go` library's internal framework of `Informers` and `Indexers`. However, this should not concern us in this section, where we only discuss the design of the system and its components at a higher level, stripped as much as possible of implementation details and technical points.

the initial data synchronization is completed. Thanks to the "distributed barrier", no matter how many Stayers fail, or how often they do so, it is guaranteed that the state of the `RicasCluster` remains consistent and that the system is resilient to $k - 1$ Agent failures before the availability of the stored data is affected, although the completion of the scale-in process may be delayed. A Stayer failing in the middle of the Scale-in Pull phase is the most straightforward kind of failure to handle.

Let us consider the possibility of a Stayer Agent failing while being in the Scale-in Barrier phase. That is to say, the Stayer has completed its own scale-in pull and flipped both its `ScaleInPending` and `SyncStatus` bits in the `RicasCluster` object, but some other Stayers have not. The Stayer in question fails while waiting for them and, at that moment, the `ScaleInFactor` and `ScaleInUnfinished` fields in the `RicasCluster` API object are still occupied by non-zero values. Upon being restarted, and before its rsync daemon is deployed to start serving incoming rsync client connections from other Agents, the Stayer flips its `SyncStatus` and `ScaleInPending` bits in the `RicasCluster` object again, to indicate that there is some data synchronization actively going on, and that the scale-in pull has to be repeated. The Stayer first enters the Init Sync phase, as usual, and starts retrieving any missing object replicas from other Agents, based on the state of the consistent hashing ring before the scale-in operation was issued. It also queues the scale-in pull anew, so as to repeat it after the initial data synchronization is over. Indeed, when it is over, the Stayer transitions from the Init Sync phase directly to the Scale-in Pull phase, as shown in Figure 3.29. The scale-in procedure is thenceforth carried out normally, as it has been already described.

Another case of failure is that of a Stayer Agent failing and being restarted after the `ScaleInFactor` has already been zeroed, but before the `ScaleInUnfinished` has been zeroed yet. In this case, the restarted Stayer Agent does not detect the pending scale-in procedure at all, because it does not really care about the Leavers anymore. Since all scale-in pulls have already been completed across all Stayers, there should be no objects stored exclusively by Leaver Agents. The state of the data across the consistent hashing ring is consistent even without having the Leavers in the `RICAS` cluster – in fact, this is the point of having zeroed the `ScaleInFactor` field and the `ScaleInPending` bit array. The failed-over Stayer enters the Init Sync phase to perform its initial data synchronization, taking into account the consistent hashing ring that comprised of the N'_s Stayers only. When this is successfully finished, it transitions to the

Ready phase, as usual. In the meantime, it is the job of the non-failed Stayers and the Leavers to scale down the backend `StatefulSet` properly before they also zero the value of the `ScaleInUnfinished` field, too.

Next, let us examine the cases that a Leaver Agent fails during a scale-in operation. Normally, the Leavers do not need to pull any data during a scale-in operation. However, most of the time, they do need to have their stored object replicas – those that were assigned to them according to the state of the consistent hashing ring before the scale-in operation was issued – available for the Stayers to retrieve. This is essential for deciding on the actions that a restarted Leaver should take, depending on the timing that it is being restarted with respect to the pending scale-in procedure.

First of all, when a Leaver is restarted from a failure, of course, it does not modify its `ScaleInPending` bit, anyway. However, it checks whether the value of the `ScaleInFactor` field in the `RicasCluster` object has been zeroed or not. If it has, the Leaver deduces that all Stayers have completed their scale-in pulls, thus no Stayer will thenceforth attempt to retrieve any of its previously assigned data. As we will examine soon, no Proxy will attempt to contact the Leaver for some incoming client request either. Therefore, the Leaver under these circumstances does not really need to complete its initial data synchronization at all. All this is decided during the Init Sync phase, and the Leaver jumps directly to the Scale-In Barrier phase, to cooperate with the rest of the Agents and Leavers to scale the backend `StatefulSet` down and zero the value of the `ScaleInUnfinished` field to signify the end of the cluster's Scale-in phase.

On the contrary, if the Leaver observes a non-zero value in the `ScaleInFactor` field, its behavior is different. In this case, it deduces that there must be some Stayers that still have not completed their scale-in pull. Therefore, the data stored across the consistent hashing ring are not yet highly available without taking into account the presence of the Leaver, thus the latter has to contribute to the resilience of the `RICAS` cluster to any additional failures. For that reason, it enters the Init Sync phase and performs its initial data synchronization based on the state of the consistent hashing ring as it was before the scale-in operation was issued, as usual, while its `SyncStatus` bit still indicates that itself is not a reliable source for the scale-in data synchronization of the Stayers yet. When the initial data synchronization is over and the `SyncStatus` bit is flipped again, the Leaver can serve rsync client connections for data retrieval by the Stayers. If, in the

meantime, the `ScaleInUnfinished` field is zeroed, which indicates that the backend `StatefulSet` has been already scaled down properly, the kubelet notifies the Leaver to gracefully shut down its operation.

3.9.4.7 The Proxies

So far we have been discussing about the various control operations of the `RICAS` cluster and we have been examining how the Agents communicate and coordinate with each other and with the assistance of the Operator. During the occurrence of any of these operations, all the data stored in the system must remain highly available, and all the coordination among the Agents that has already been shown aims exactly at that feature: high availability through a hard guarantee of fault tolerance. However, when discussing about the availability of the data towards the clients, we should never forget that the frontend has an equally important role to play in that, since the first and the last component of the system to receive any client request and its corresponding response by the system is always a Proxy.

The operation of the Proxy is affected by the scaling operations on the `RICAS` cluster. When the cluster is either scaled out or scaled in, the topology of the virtual nodes on the consistent hashing ring is modified. As it should be self-evident by now, when that happens, a lot of object keys are designated to different Agents than the ones they were assigned to prior the scaling operation. This is crucial for the functionality of the Proxies: it is them that are actually responsible for forwarding the incoming client requests to the appropriate Agents on the backend, thus for orchestrating the whole procedure of the assignment of objects to ring nodes according to the state of the consistent hashing ring each time. Thereby, although the Proxies are not involved in the decisions that are being made about the course of the scaling operations themselves, they must always be aware of latest status of the `RICAS` cluster for the latter to be operating soundly.

To begin with, we have to recall a feature of the system, supported in both `MICAS` and `RICAS`, and implemented on the frontend tier. We refer to it as "**adjustable read-after-create consistency level** at **per-object granularity**", and it has been explained in the context of `MICAS` in subsection 3.8.5. Basically, every client `Create` request may

include an integer c in the range $[1,\ k]$, where k is the configured value of the replication factor, and c is the *consistency level* of the creation request for the particular object. A Proxy processes the value of c received in the `Create` request by the client, and through it, the Proxy is instructed to consider the creation of the object successful even if it is confirmed for only c out of k replicas that they were stored locally by an Agent in a durable manner within the allowed duration determined by the configured time-out for the communication between the frontend and the backend tiers. When $c < k$, effectively, the Proxies do not need to wait to receive confirmations for the creation of the object replica from all k Agents, and this usually leads to lower latency in handling the incoming requests, as observed by the clients, which is sometimes valued over the strong guarantee that all k replicas are available after the creation of the object – the strong **read-after-create consistency**.

During a scaling operation, the behavior of the Proxies is slightly altered than what it has been described so far and that holded for both `RICAS` and `MICAS`. For the duration of the scaling operation, regardless of whether it is a scale-out or a scale-in, the `RICAS` Proxy maintains both versions of the consistent hashing ring: both the old one, i.e. the state that the ring was in before the scaling operation was issued, and the new one, i.e. the state that the ring will be in after the scaling operation has finished. To retain high availability of the `RICAS` cluster and its stored data, clients' `Create` and `Read` requests are handled taking into account both the old and the new state of the consistent hashing ring. It is underlined that although Proxies' behavior during scaling operations is different than during the cluster's normal operation, this differentiated behavior is actually the same for both the scale-out and the scale-in operations.

With the handling of `Create` requests being both the most significant alteration and also what defines, more or less, how the handling of the `Read` requests is altered as well, we start off by examining in detail this case first. When a `Create` request arrives at a Proxy from a client, the Proxy calculates its hash digest, which is thenceforth considered to be the key of the new object, as usual. Assuming that we are in the middle of a scaling operation, the `RICAS` Proxy searches both the old and the new consistent hashing ring to find two sets of Agents that should own a replica for the object – one for each state of the ring. Of course, if the value of the scaling factor (i.e. the `ScaleOutFactor` or the `ScaleInFactor` if the scaling operation is a scale-out or a scale-in, respectively) is lesser than the value of the replication factor k, those two sets of Agents

should not be disjoint, but the course of the procedure that we describe is the same, irrespectively of that. At this point, the Proxy forwards the `Create` request to all these replica owners (their number should be anywhere in the range $[k,\ 2 \cdot k]$). To consider the creation of the object successful, the Proxy awaits confirmations from all k Agents that should own a replica of the object according to the old state of the ring, and from at least c out of the k Agents that own a replica of the object according to the new state of the ring.

To understand why this is necessary, let us examine and analyze a concrete example. Assume that a deployed `RICAS` cluster is in the middle of a scale-out operation. The – previously – N_s Agents (the Elders) are joined by M fresh additional Agents (the Juniors), to form a cluster of N'_s Agents in total at the end of the scale-out operation. At this point, as it was described in §3.9.4.4 in detail, the recently spawned Juniors must be performing their initial data synchronization, by traversing the consistent hashing ring as it was before the scale-out operation was issued in order to figure out which data they should be storing replicas of.

- If the Proxy forwarded the `Create` request *only* to the Agents that are owners of a replica of the object according to the *old* state of the consistent hashing ring, then some replicas, or maybe even all replicas for some objects, may be entirely lost. In the ring traversal performed by a Junior for the purpose of its own data synchronization, let us consider a ring segment where the objects are owned by that Junior (i.e. a ring segment formed by one of the Junior's virtual nodes), and which had already been synchronized when the client's `Create` request arrived at the Agent from the Proxy. Let us also assume that the key of the new object falls right into this ring segment. It becomes evident that, by letting only the "old replica owners" – which, by the way, are all Elder Agents, by definition – to store the new object, there is no way for that Junior to ever acquire that object anymore. As a matter of a fact, if the scaling factor is greater than or equal to the replication factor, and this happens to all Juniors (which is actually not at all unlikely), when the scale-out operation is over and the Elders also finish their Garbage Collection phase, it is possible that not even a single replica of the object is stored across the cluster in the end, whereas the client will have been incorrectly notified of a successful `Create` operation.

- If the Proxy forwarded the `Create` request *only* to the Agents that are owners of a replica of the object according to the *new* state of the consistent hashing ring, again, either some replicas of some objects, or maybe even every single replica of some objects, may be entirely lost. Let us consider such a case of a `Create` request that has been successfully handled by a Junior Agent while the latter is in the middle of its initial data synchronization, and that a little later – but before the data synchronization is over – the Junior fails and gets restarted. Since the Junior performs its initial data synchronization by pulling data only based on the old state of the consistent hashing ring, if the Elder Agent that will be contacted for the transfer of all objects under the same virtual node as the object of the `Create` request in question, is not one of the "new replica owners" for that object, too, then the Junior will not retrieve the object again, even though it has already responded to the Proxy that the object would be thenceforth stored locally to it. If the same thing happens to multiple Juniors, and we are a little "unfortunate" with the topology of the consistent hashing ring and the assignment of the objects based on it, it is possible that some of the objects created during the scale-out operation end up being completely lost.

The situation is quite similar in the case of a scale-in, where M of the initial N_s Agents are being removed from the `RICAS` cluster, leaving it with N'_s Agents.

- If the Proxy forwarded the `Create` request *only* to the Agents that are owners of a replica of the object according to the *old* state of the consistent hashing ring, individual replicas of some objects, and possibly even all replicas of some objects, may be completely lost. In the ring traversal performed by a Stayer for the purpose of its own scale-in pull, let us consider a ring segment, the objects of which are to be owned by that Stayer after the scale-in operation is over (i.e. it is a ring segment formed by one of the Stayer's virtual nodes in the new ring). Most of the "old replica owners" of a newly created object could be Leaver Agents (or even all of them could be Leavers if the scaling factor is greater than or equal to the replication factor). Let us also assume that this ring segment had already been synchronized when the client's `Create` request arrived at the Agent from the Proxy. In this case, it is possible that the Stayer never retrieves a replica of the object in time. In fact, if all "old replica owners" turn out to be Leaver Agents,

the new object will be soon entirely lost when the scale-in phase is over and the Leavers are terminated. However, the Proxy (and probably the client, too) has already been notified that the object is successfully stored in the system, causing inconsistency.

- If the Proxy forwarded the `Create` request *only* to the Agents that are owners of a replica of the object according to the *new* state of the consistent hashing ring, individual replicas of some objects or even whole objects could be lost, too. Let us consider such a case of a `Create` request that has been successfully handled by a Stayer Agent while the latter is in the middle of its scale-in pull, and that a little later – but before the data synchronization is over – the Stayer fails and gets restarted. As explained in §3.9.4.6, the Stayer, first, performs its initial data synchronization based on the old state of the consistent hashing ring, the one that includes the Leavers, to retrieve any replicas of objects that it should own anyway before the scale-in operation had been issued. It is possible that the object in question was not assigned to this particular Stayer prior the scale-in, thus it might not be retrievable through the initial data synchronization procedure at all. In this case, this object could only be retrieved from some Leaver during the scale-in pull pending after the initial data synchronization is over. If the Leavers, which can only be "old replica owners" by definition, do not have this object stored, it may not be ever available at all, even though the client would be falsely notified that the object is stored in the system.

In general, the initial data synchronization of an Agent during a scaling operation is always performed based on the old state of the consistent hashing ring, i.e. based on its state before the scaling operation was issued. A restarted Agent that failed during a scaling operation relies heavily on this synchronization for its stored data, hence the new data must be written in such a way so that the old state of the ring is reliable. On the other hand, no Agent can rely exclusively on the data synchronization procedure of the scaling operation for the most recently created data, hence the new state of the ring must also be taken into account when new data are stored in the system while a scaling operation is in place.

Now that we understand why we need to make sure that both the old and the new replica owners store the new objects that are being created during the scale-out oper-

ation, another reasonable question pops up. Why store it to k replica owners of the old state of the ring and to at least c replica owners of the new ring? Let us consider the example of the scale-out operation, again. The object replicas are required to be locally stored at all k Agents that should own them according to the *old* state of the consistent hashing ring, because in the case that a Junior fails during the scale-out, it is not known which Elder Agent will it contact when it is restarted to retrieve the associated data of the virtual node from. Hence, to ensure that this Junior indeed has this object stored in the end, all of the Elders should own the object. On the other hand, the replicas of the object are required to be locally stored only at c out of the k Agents that should own them according to the *new* state of the consistent hashing ring, in order to consider the creation successful. This is in line with the policy of the adjustable consistency level that is provided – it is what would happen anyway if the scale-out operation had been already finished.

A perspicacious reader may have already spotted the flaw that has been unveiled after the above discussion, concerning the adjustable consistency level at per-object granularity. Setting the value of the consistency level c to be lesser than the value of the replication factor k for some objects, effectively allows some Agents to entirely ignore the new object. This ignorance can then be propagated by the means of the data synchronization when a perfectly consistent (regarding its stored data) Agent fails and gets restarted over some other physical node. It begins its initial data synchronization to retrieve all the data it should own, by contacting the other Agents that own replicas of the objects that lie in the ring segments that its own virtual nodes form on the consistent hashing ring. If these other Agents that are being contacted had not created some of those objects for some reason, legitimately, having configured $c < k$, then the objects may never reach the failed-over Agent again.

Gradually, this behavior may even lead to data loss. Of course, this is similar to the risk that the client was willing to take when the consistency level was set for the object. To mitigate that risk without totally abandoning the consistency level feature, there are two actions that can be made. The first, which is also the most sophisticated solution, is to introduce a "*repairing mechanism*" to periodically synchronize the data among the Agents, even when no failures or scaling operations take place. It could be designed as an expansion of the system, by adding another `RICAS` cluster phase, as well as one

or more phases of for the RICAS Agents, as shown in Figure 3.28 and Figure 3.29, respectively. The second solution, which is kind of rough, yet effective, is to slightly change the way the data synchronization takes place. Instead of contacting only one Agent per ring segment to retrieve its objects, the procedure could be modified so that an Agent contacts every other Agent that is responsible to store replicas of the objects in that ring segment. The same would happen for every ring segment the objects of which need to be synchronized. In this way, the Agent would effectively retrieve the union of the sets of object replicas stored by all of the Agents that are responsible for every segment, hence the ignorance of some Agents about some of the objects would not be propagated.

Having settled on the exact way that the Create requests are handled during a scaling operation, it is time to discuss about the handling of the incoming Read requests. This is quite straightforward, now. Since the system needs to make the best effort in serving stored data, and knowing that during scaling operations the new data are stored by the replica owners of the corresponding ring segment based on both the old and the new states of the consistent hashing ring, all of them are contacted by the Proxy to respond to a Read request. The Proxy searches the object's key in both the old and the new state of the consistent hashing ring, and concludes to two sets of Agents, each of size k. Then, it forwards the Read request to all Agents in the union of the two sets, a set of size in the range $[k,\ 2 \cdot k]$, as the two sets may not be disjoint. The Proxy responds to the client when one of these occur:

- it receives the corresponding stored blob from one of the Agents,
- all Agents report that they do not have the object stored locally,
- the time duration specified by the configured time-out for the communication between the frontend and the backend layers is over.

3.9.4.8 Summary

Rather than presenting the data flow during the handling of Create and Read requests in detail as we did in the case of MICAS, which should pretty much self-evident by now, we conclude the analysis of RICAS by presenting a summary of the phases that a RICAS

Agent can find itself in, as well as all possible transitions from one phase to another. We do that by enumerating all those transitions as annotated in Figure 3.31 (which is the same as Figure 3.29, but also includes the annotations) and briefly commenting on each one of them.

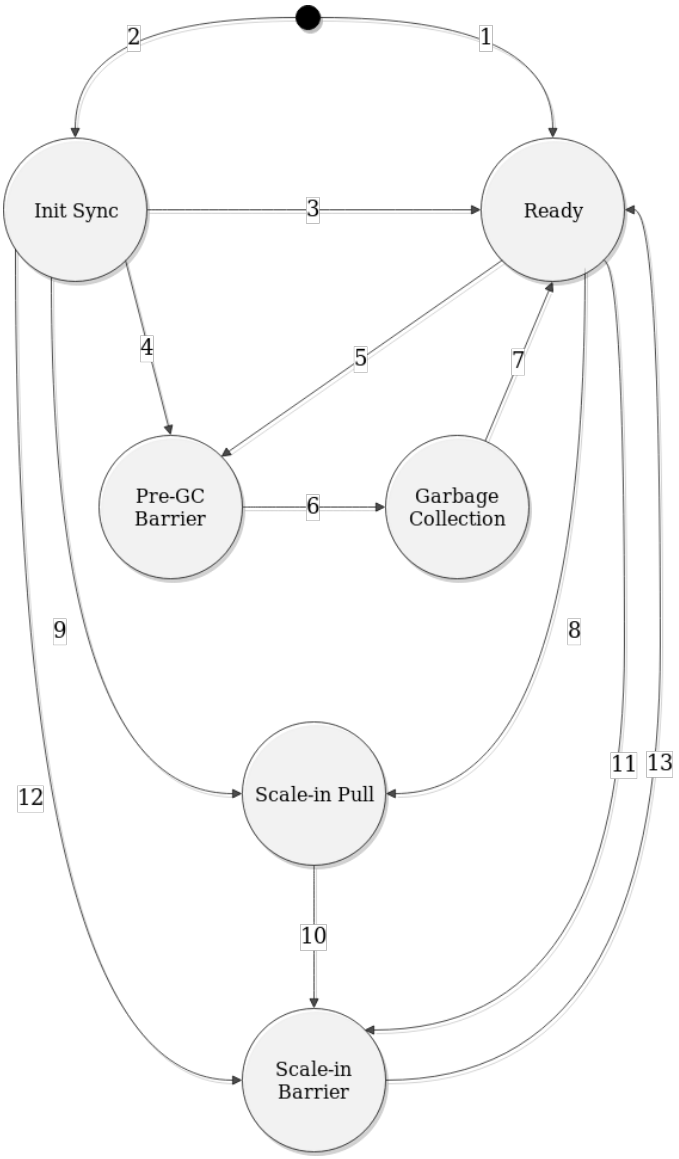


Figure 3.31: *Same state diagram as in Figure 3.29, properly annotated to summarize phase transitions.*

1. *start* $\longrightarrow$ **Ready phase**

   The Agent enters the Ready phase for the first time since its deployment, after the `RICAS` cluster has just transitioned from the Bootstrapping phase to the Ready phase, as per Figure 3.28.

2. *start* ⟶ **Init Sync phase**
   The Agent enters the Init Sync phase upon booting under any of these circumstances:
   - after being restarted from a previous failure, while the `RICAS` cluster may be in any phase;
   - when it is deployed for the first time as a Junior, as a result of the latest scale-out operation that was issued, while the `RICAS` cluster is in the Scale-out phase.

3. **Init Sync phase ⟶ Ready phase**
   The Agent has just completed its initial data synchronization, and the `RICAS` cluster is not in the middle of a scaling operation.

4. **Init Sync phase ⟶ Pre-GC Barrier phase**
   The Agent has completed its initial data synchronization very recently, the `RICAS` cluster is in end of the Scale-out phase and about to transition to the Garbage Collection phase, and one of these holds true:
   - the Agent is a restarted Elder that failed during this scale-out operation;
   - the Agent is one of the Juniors that joined the `RICAS` cluster as a result of this scale-out operation.

5. **Ready phase ⟶ Pre-GC Barrier phase**
   The `RICAS` cluster is in the end of the Scale-out phase and about to transition to the Garbage Collection phase, and the Agent has been an Elder which did not fail recently – or did not fail at all – for the duration of this scale-out

6. **Pre-GC Barrier phase ⟶ Garbage Collection phase**
   The Agent has just observed that all the other Agents in the `RICAS` cluster have reached the "distributed barrier" prior to the Garbage Collection phase, so it can proceed with its own local garbage collection of the objects that are not required to be stored by it anymore.

7. **Garbage Collection phase ⟶ Ready phase**
   The Agent has just completed its local garbage collection and has let all the other Agents know about it by properly modifying the associated `RicasCluster` API

object. The RICAS cluster has been in the Garbage Collection phase, but it may transition back to the Ready phase if this Agent is the last one to be doing this transition.

8. **Ready phase ⟶ Scale-in Pull phase**
   The Agent – a normally operating one, not one recovering from some previous failure – has just observed that there is a pending scale-in, during which itself is a Stayer. Therefore, it transitions from the Ready phase to the Scale-in Pull phase to begin its scale-in data synchronization.

9. **Init Sync phase ⟶ Scale-in Pull phase**
   The Agent has just completed its initial data synchronization and observes that there is a pending scale-in, during which itself is a Stayer. The failure, due to which the Agent was restarted, may have occurred either before the scale-in operation was issued, or after that. The Stayer Agent transitions from the Init Sync phase to the Scale-in Pull phase to begin its scale-in pull in order to retrieve its newly assigned objects, immediately after having retrieved the data that had been already designated to it based on the old state of the consistent hashing ring (i.e. before the scale-in operation was issued). The RICAS cluster is in the Scale-in phase, according to Figure 3.28.

10. **Scale-in Pull phase ⟶ Scale-in Barrier phase**
    The Agent has just completed its scale-in data synchronization – of course, while the RICAS cluster is in the Scale-in phase – and proceeds to the Scale-in Barrier phase, waiting for the rest of the Agents in the cluster to finish their own scale-in pulls as well.

11. **Ready phase ⟶ Scale-in Barrier phase**
    The Agent – a normally operating one, not one recovering from some previous failure – has just observed that the RICAS cluster transitioned to the Scale-in phase, and that itself is a Leaver. Contrary to the case of the Stayers (which transition to the Scale-in Pull phase), the Leavers do not have any new data to pull, since they are the ones that are being removed from the cluster. However, they must wait until all Stayers have completed their scale-in data synchronization, to retain the high availability of the deployed system regarding its stored objects.

Thereby, the Leaver Agent transitions to the Scale-in Barrier phase, and it stays there serving both client requests from the Proxies and data synchronization requests from the Stayer Agents.

12. **Init Sync phase ⟶ Scale-in Barrier phase**
    The Agent is a Leaver that has been recently restarted from a previous failure, which may have occurred either while the `RICAS` cluster is in the Scale-in phase, or earlier. Unless all Stayers have completed their scale-in data synchronization pulls, the Leaver has just completed its initial data synchronization based upon the old state of the consistent hashing ring (i.e. the one before the scale-in operation was issued), and it proceeds to the Scale-in Barrier phase to cooperate with the Stayers, serving both them and the Proxies. If every Stayer has completed its scale-in data synchronization pull, the Leaver skips its own initial data synchronization, since it is about to be terminated, and skips directly to the Scale-in Barrier phase to join the other Agents in scaling down the backend `StatefulSet`.

13. **Scale-in Barrier phase ⟶ Ready phase**
    The Agent has made sure that the backend `StatefulSet` has been scaled down properly and that the Leavers are about to terminate. The `RICAS` cluster has either transitioned to the Ready phase already, or is about to do so. If the Agent in question is a Stayer, it proceeds to the Ready phase to continue operating normally. Otherwise, if it is a Leaver, it proceeds to the Ready phase waiting to be notified by its node's kubelet to begin the graceful termination of its component entities and its operation overall.

4 Implementation

In this chapter we intend to shed some light on a number of sectors of the implementation of the two systems, and especially on RICAS. After presenting some of the theoretical background, as well as the major design rationale and overall architecture of both systems, there are several aspects yet undiscussed. These include major pieces of each system, such as the storage egine used on the backend tier and the aforementioned rsync-based object synchronization mechanism, but also include technical details, such as the hash function that was finally selected (based on the analysis of section 3.5) and the final form of MICAS' and RICAS' external API.

The process of the implementation of each system took place in multiple iterations, much like their design. This process included a complete rewrite of both MICAS and RICAS in a different language, which has actually been liberating for the reasoning of their structure, thanks to the variety of libraries and tools that are available in each language. At first, both systems were developed in the Python programming language. At this point, the only the – naive – RESTful external API was developed, leveraging raw HTTP messages containing Base64 encoded, JSON serialized information. This version of MICAS, thanks to the overall simplicity of its design, is fully functional. However, this is not the case for RICAS, which supports only a subset of the operations described in chapter 3 in its Python version.

RICAS' functionality, being somewhat more complex than MICAS', relies heavily on the *controller pattern*. This pattern, although it is increasingly popular in Kubernetes' ecosystem, is most straightforwardly implemented using the `informers` "framework", which is only included in Kubernetes' client library for the Go programming language

– the language that Kubernetes itself is developed in, which is known as `client-go`. The use of `informers`, together with the simplicity and the built-in concurrency features of Go, in conjunction with several adjustments and fixes to our own logic, of course, led to the final version of `RICAS`, which is developed in Go. It also exposes the external gRPC API that was described in section 3.2. `MICAS` has also been rewritten in Go, so as to take advantage of the imporoved performance and concurrency features, too. In its final version it exposes a external gRPC API as well.

Regarding the following sections, it is noted that, unless otherwise explicitly stated, when referring to the implementation of either `MICAS` or `RICAS`, we refer to their final versions, in Go. It is also remarked that hardly any code segment is demonstrated or analyzed; with a total size of the codebases spanning over 15000 SLOC[1], such an effort of elaborate presentation would be, if anything, impractical, and inevitably incomplete. However, the complete source code is hosted on GitHub, and can be found at:

- `https://github.com/ckatsak/pyntainerdb`, for the Python version of `MICAS`;
- `https://github.com/ckatsak/hexycas`, for the Go version of `MICAS`;
- `https://github.com/ckatsak/pyntainerdb-ring`, for the Python version of `RICAS`;
- `https://github.com/ckatsak/mhacaretes`, for the Go version of `RICAS`.

4.1 Storage Engine

At various points of the presentation of both `MICAS` and `RICAS` in the previous chapter, we referred to an underlying storage engine.

In general, a storage engine, or database engine, is the underlying software component that a storage system uses to create, read, update and delete data from the database. Examples of systems that are often used as storage engines for other storage systems are MyISAM[188], InnoDB[189], WiredTiger[137], LevelDB[135] and RocksDB[136]. Usually, storage engines include their own API, separately from the one implemented by

[1]Source lines of code.

the storage system for interaction with the user, and which is used by the storage system to achieve its goals.

This is the case for our systems, too. In both the design and the implementation of MICAS and RICAS, an attempt has been made to decouple the storage engine from the rest of the functionality of the system. Our aim is to make it possible and easy to plug different storage engines with minimal modifications to the other components of the system. The storage engine is always a part of the Agent, since the Agents are the only components that actually must store data.

However, this thesis does not mean to focus on the lower storage components of the system. It rather is an attempt to explore the functionality and the features of Docker and Kubernetes and how they can be beneficial to the design and the implementation of a complete distributed storage system. Therefore, instead of selecting and focusing on a particular storage engine, we choose to implement ourselves an extremely naive one, which merely fulfills our basic requirements, letting us concentrate on other aspects of the system.

4.1.1 The Storage Backend Abstraction

To achieve the goal described below, in RICAS, the storage engine is abstracted away from the rest of the system. For that reason, the `storage.Backend` and `list.Handle` abstractions are defined as Go interfaces:

```
1 type Backend interface {
2     Create(key, value []byte) error
3     Read(key []byte) (value []byte, err error)
4     List(start, end []byte) list.Handle
5 }
```

Listing 4.1: *The definition of the* `storage.Backend` *interface, used by the* `BlobServer` *component-thread to perform the required subset of the CRUDL operations.*

```
1 type Handle interface {
```

```
2     NextChunk() (keys [][]byte, err error)
3 }
```

Listing 4.2: *The definition of the* `list.Handle` *interface that appears in the definition of the* `storage.Backend` *interface, and is used by the* `BlobServer` *component-thread to perform the* `List-Range` *and* `List` *operations in earlier versions of the systems.*

The component of the system that is responsible for issuing the required subset of the CRUDL operations [2] (this component-thread is the `BlobServer`, as it will be revealed later) only knows about the presence of this basic functionality of the storage abstraction. The actual underlying implementation can be anything, depending on the storage engine being used. Thus, a certain degree of freedom is given to plug a number of different storage engines depending on the needs of the deployment each time, while keeping the modifications to the rest of the implementation of the system as few as possible.

However, the exact way that the data are stored in the system does not only affect the component that performs the CRUDL operations on behalf of the client requests. It affects every operation that is performed on the data, including those that are initiated as a result of control operations. For instance, the initial data synchronization and the scale-in data synchronization depend on it, which are both performed by the Agent's `Syncer` component-thread. So does the local garbage collection, performed by the Agent's `GarbageCollector` component-thread. Therefore, there are two additional abstractions that have to be defined: the `syncer.backend` and the `storage.gcBackend`.

```
1 type backend interface {
2     Pull(*lfchring.VirtualNode, lfchring.Node, *lfchring.HashRing)
          bool
3 }
```

Listing 4.3: *The definition of the* `syncer.backend` *interface, used by the* `Syncer` *component-thread to perform the initial data synchronization and scale-in data synchronization pulls.*

[2] Namely, only `Create` and `Read` (and also `List-Range` and `List` in earlier versions of the systems.)

```
type gcBackend interface {
    GarbageCollect(ringSize int)
}
```

Listing 4.4: *The definition of the* `storage.gcBackend` *interface, used by the* `GarbageCollector` *component-thread to perform the local garbage collection.*

4.1.2 Our Own Naive Implementation

Our simplistic storage engine relies entirely on the underlying filesystem. The data are stored in the form of objects and each replica of such an object corresponds to a single file on the underlying filesystem. New objects are created by creating new files and writing the data to them, e.g. using the `write(2)` system call. Later, these objects can be read by simply reading the new files and returning their content, e.g. using the `read(2)` system call. The read operations in our jejune storage engine are also benefited by the presence of the page cache in the Linux kernel. The page cache accelerates some accesses to files by using otherwise unused areas of memory as a cache for pages that normally live in secondary storage, i.e. hard drives. This happens for both pages of files (objects in our case), as well as for the directories that the files may be in (in our case they always are, as we will explain soon).

At this point, it is important to make a note about the expected performance of the system. Filesystems and databases, each aims to solve different a problem by providing different abstractions and features. Plainly using the filesystem as the storage engine of a distributed storage system, like `MICAS` and `RICAS`, is expected to be ineffective in terms of performance, for various reasons. One of the main reasons is relevant to durability, and will be presented in detail at the end of this section.

4.1.3 Index

Before diving into the details of the durability requirements of our system, we ought to examine another important supported feature. As explained in section 3.1, the objects are stored in the system in one totally *flat namespace*. This means that there are no ad-

ditional buckets, containers or collections of objects, and no way for clients to enforce a certain data schema. Although this is true for the data model of the system as it is exposed to the clients, our naive storage engine, in both `MICAS` and `RICAS`, internally uses an index to store and retrieve objects efficiently.

Our index is actually heavily based on the indexing mechanism present in all traditional filesystems: the directories. In general, according to the idea of the "*common file model*" behind Linux kernel's *VFS*[3], each directory can usually be regarded as a file that contains a list of files and other directories [190] [4]. Since the number of objects that have to be stored by a single Agent can get quite large, keeping all of them under a single directory entry can be costly, even for efficient filesystem implementations. Therefore, we decide to divide them into groups, where each group is actually a directory.

First of all, all these groups should be populated by objects as evenly as possible. To this end, assuming the hash function that has been chosen is in accordance with the analysis that was presented in section 3.5, the logic of the grouping is based on the keys of the objects themselves, the hash digests. Secondly, when the number of stored objects is very large, a hierarchical grouping method might be even more beneficial. Combining these two points, we end up using a directory hierarchy; a tree of directories and their subdirectories, with a **number of levels that is configurable** at the time of the deployment, and a **fanout of sixteen** subdirectories.

In our naive storage engine, every stored object is represented as a file. The content of the file is the actual data, the blob that needs to be stored, whereas its name is the object's key, the hash digest, represented in hexadecimal form. We use the hexadecimal digits that are present in the name of file (the object's key), from the most to the least significant one, to decide on which group each object should be in. In our directory hierarchy, there is a top level, the root of the hierarchy, and beyond that, each level comprises 16 subdirectories, one for each hexadecimal digit in the range from `0x0` to `0xf`. Starting from the most significant hexadecimal digit, the i-th digit indicates the subdirectory that the file should be stored in, in the i-th level.

Figure 4.1 is an attempt to illustrate an example of the directory hierarchy that consti-

[3] Virtual File System.

[4] Of course, the actual implementation may differ, mainly for reasons of performance and efficiency.

tutes our system's naive internal index, in an example where the number of levels has been set to four. Note that not all subdirectories are shown in the figure, for practical reasons.

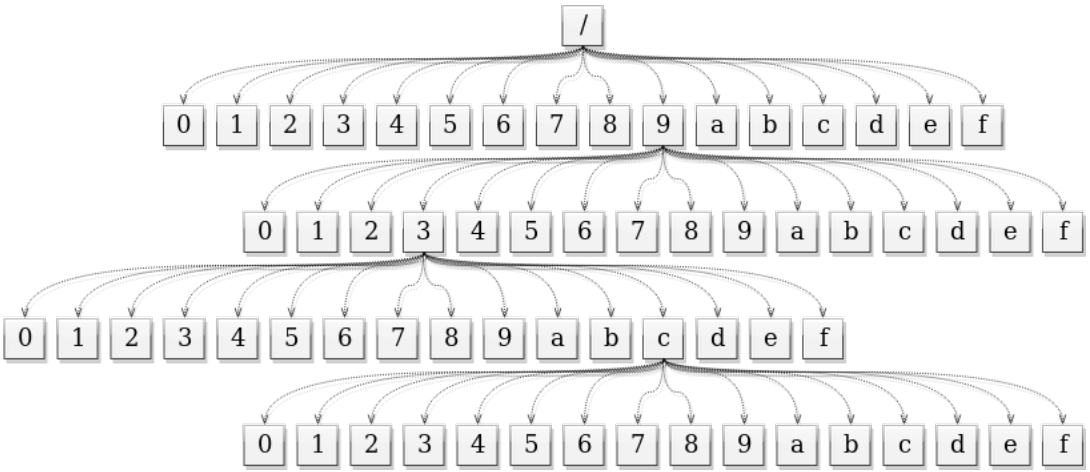


Figure 4.1: *An attempt to illustrate an example of the directory hierarchy that constitutes our system's naive internal index. In this example, the* ***index height*** *has been set to four. Not all subdirectories are shown in the figure, for practical reasons.*

This index is considered a part of our simplistic storage engine implementation, thus it is present in both `MICAS` and `RICAS`. As we already stated, the number of levels that are present in the index is configurable at the time of the deployment of the system. For now, its reconfiguration after the system has been deployed is not supported, so the administrator has to make a choice based on the order of magnitude for the maximum number of objects that are expected to be stored. We call this configurable number of levels the **index height**, since it is analogous to the height of the tree of subdirectories that is formed by this hierarchy. The fanout of this tree is always at most sixteen, because of the sixteen hexadecimal digits that it is possible to exist in each level. Therefore, the maximum number of groups of objects that are formed by configuring the index height, i.e. the number of leaves in the index tree, at the time of the deployment of the system, is static and can be calculated by (4.1).

$$\#\, index\ leaves \;=\; 16^{index\ height} \tag{4.1}$$

Currently, the maximum value for the index height is eight, which, according to (4.1), leads to $16^8 = 2^{32} \approx 4 \times 10^9$ leaf-subdirectories. These are, anyway, more than a lot for the expected needs of any `MICAS` or `RICAS` deployment. Moreover, the larger the

number of directories, the greater is the contention in kernel's dentry cache; hence, such a huge number of directories would not necessarily be rewarding in terms of efficiency and performance. Although the actual value of the index height may differ depending on the needs of the deployment and, of course, on the underlying filesystem, in practice, no more than three to four levels are expected to ever be needed.

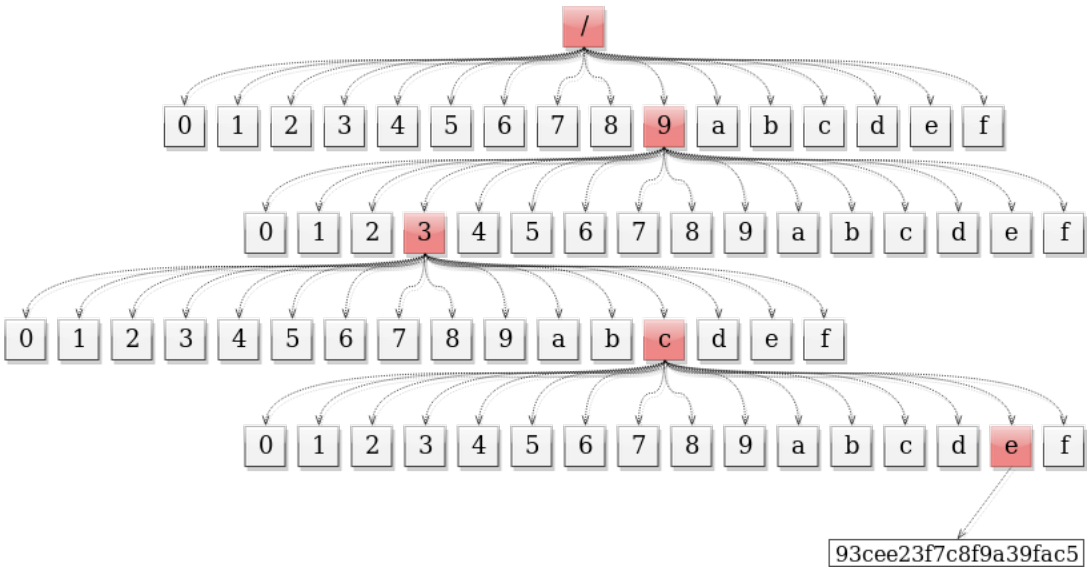


Figure 4.2: *An example of locating the stored object (file)* `93cee23f7c8f9a39fac5` *in the directory hierarchy of our system's naive index of Figure 4.1. The index height has been set to four. Not all subdirectories are shown in the figure, for practical reasons, and the marked subdirectories form the object's final path.*

Figure 4.2 depicts an example of locating a stored object (file), let its key (hash digest, file's name) be `93cee23f7c8f9a39fac5`, using the four-level index of Figure 4.1. We examine the four most significant hexadecimal digits, in the order of their significance: `9`, `3`, `c` and `e`. Starting at the root of the index, we proceed by matching the i-th digit to the i-th level of the directory hierarchy, and we conclude that the object `93cee23f7c8f9a39fac5` should be stored in directory `/9/3/c/e/`. Note that the root of the index may not correspond to the root of the filesystem. Therefore, assuming that the root of the index is in the directory `$INDEX_ROOT`, the path of the object in the filesystem is actually `$INDEX_ROOT/9/3/c/e/93cee23f7c8f9a39fac5`.

Last, we have to make a remark about the creation of the directories of the index hierarchy. At first glance, all of them could be created at the time of the deployment of the Agents. However, as we have examined before, in `RICAS`, each Agent is assigned specific segments of the consistent hashing ring, the ones that correspond to its own

virtual nodes. Thereby, many of the directories might be completely useless for the `RICAS` Agents to create. Similarly, although less noticeably, some directories might end up being useless for the `MICAS` Agents, too. For that reason, our naive storage engine creates the directories of the index **lazily** exactly when they are needed for the first time. Likewise, they are removed during local garbage collection procedure when they are no longer needed.

4.1.4 Concurrency & Atomicity

As we have already stated earlier, the incoming requests from the Proxies are handled concurrently, by multiple threads [5]. If multiple incoming requests refer to the same object, it is possible that two or more threads attempt to operate on the same file, thus inevitably stumbling upon race conditions. For that reason, we have to make sure that our simplistic storage engine is able to handle, basically, the creation of new objects (i.e. files) in an **atomic** manner.

Since our system does not support `Update` operations at all, no race condition can occur due to some modification of the data; there is no notion of such "conflict" or "overwriting" at all. Any possible race condition can only be relevant to some `Create` operation. In fact, there are two types of such race conditions that may occur; in their simplest forms they may appear as such:

- two `Create` operations for the same blob are handled concurrently by two different threads;
- a `Create` operation and a `Read` operation for the same blob are handled concurrently by two different threads.

In both cases, opening a file and writing data to it, or reading data from it, e.g. using the `open(2)`, `write(2)` and `read(2)` system calls, is not guaranteed to be atomic by the Linux kernel; these are procedures that have to be completed in multiple steps by each thread. Therefore, each thread might be intercepted at any point during this

[5] Technically, each request is handled in a separate goroutine rather than an operating system thread. However, goroutines still can be considered logical threads as far as the design and the implementation of the system is concerned. Moreover, they can be indeed scheduled to run on different operating system threads.

procedure by the other thread, ultimately leading to inconsistent results, which might even go unnoticed.

This complication can be solved by ensuring that the creation of the object is atomic. To achieve atomicity, we use a common "file locking" technique that is based on the `link(2)` and `unlink(2)` system calls [191]. The `link(oldpath, newpath)` system call creates a new hard link called `newpath` pointing to the same inode as `oldpath` and increases the link count by one. The call fails with the error code `EEXIST` if `newpath` already exists, making this a useful mechanism for locking a file amongst threads or processes that can all agree upon the name `newpath` [192].

The technique is based on the fact that the `link(2)` system call fails if the new link already exists. The implemented approach is to have each thread create a temporary file with a unique name, and have it write the new blob on it. Then, the temporary file is linked to the agreed-upon pathname that consists of the directories' path of the index and finally the hash digest of the object (its key). If the `link(2)` call succeeds, the new file has been created "atomically" – thus, so has the new object in our storage engine. The thread unlinks the temporary file and responds to the Proxy properly. If the `link(2)` call fails, the thread has failed to create the file atomically. Nevertheless, it is possible that another thread managed to do so. Thereby, the thread must check anew whether the final pathname exists (e.g. using the `stat(2)` system call), and respond to the Proxy accordingly, after unlinking its own temporary file.

4.1.5 Durability

In the description so far, we have deliberately omitted any reference to the durability of the objects that are being stored in the system through our naive storage engine. This is the subject of this subsection.

Under the usual circumstances, opening a file and writing data to it, using the `open(2)` and `write(2)` system calls, does not by itself guarantee that the data have actually been written to the underlying storage device. Unless some special configuration is in place, the `write(2)` call succeeds by the time the data are written to the kernel's page cache. However, an untimely node failure may prevent the kernel's page cache from being flushed to the secondary storage media (i.e. the disk). Unless handled with special

care, this situation may lead to catastrophic data inconsistency. For instance, the client may have been notified that the object has been successfully created before the data are actually flushed to the disk, and a crash prevents the system from doing so, hence leading to data loss.

To this end, the kernel provides a couple of system calls, namely `fsync(2)` and `fdatasync(2)`, which aim to resolve situations like this [6]. In general, the functionality of both of them is to block the calling thread until the data they refer to have definitely been successfully flushed to the filesystem's underlying storage device. In particular, the `fsync(2)` system call transfers all modified pages of the file it refers to, to the permanent storage device, including writing through or flushing any disk caches present, so that all changed information can be retrieved even after the system crashes or reboots, and it also flushes metadata information associated with the file. The `fdatasync(2)` system call is similar to `fsync(2)`, with the exception that it does not flush modified metadata, unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled, aiming to reduce disk activity for applications that do not require all metadata to be synchronized with the disk [193]. The `fdatasync(2)` system call is obviously not applicable in our case, because all objects are created once, and never updated; thus, the metadata information need to be written once, and once only, anyway. Thereby, we rather focus on `fsync(2)`.

The durability step that was omitted is actually the sound use of `fsync(2)`. After the object has been successfully created atomically, as described in the previous subsection, the storage engine has to ensure that the object is also flushed to the disk before allowing the Agent to respond to the Proxy. This is achieved by forcing each thread to call `fsync(2)` on the file descriptor of its own temporary file after the blob has been written to it, and before the `link(2)` call is issued. In this way, we rest assured that when the `link(2)` succeeds and the object is created atomically, the data have already been stored persistently, instead of relying on a future successful `fsync(2)` call.

However, a successful `fsync(2)` call for the newly created file does not suffice to ensure the durability of the object on its own. The kernel structure that is responsible for associating files' inodes with their pathnames, is the dentry. In our case, in order for the new file's inode to be accessible through its pathname (in our case, the path in the

[6] To be pedantic, the kernel also provides the `aio_fsync(2)` system call for a similar purpose, but we will not refer to it at all in this thesis.

directory hierarchy of our index, plus the object's hash digest), the associated dentry has to be flushed to the disk, too, as a directory entry. Unfortunately, the `fsync(2)` call on the file descriptor of the newly created file does not necessarily ensure that the entry in the directory containing the file has also reached the disk. For that, another explicit `fsync(2)` call on a file descriptor for that directory is also required [193].

The cumbersome aspect of using `fsync(2)` is that it is notoriously slow in terms of performance. This is pretty much expected due to the latency of disk I/O. Reading data from and writing data to the disk has always been a typical cause of bottleneck for storage systems, even though the quality of the hardware continues to improve and despite the advent of the much faster solid state drives in the datacenters, which have largely replaced the traditional rotating hard disks. The latency of such I/O operations is known to be by many orders of magnitude greater than the latency introduced by accessing the CPU's registers, the caches and even the main memory; in fact, it is often comparable to the latency of network I/O operations.

Using this simplistic storage engine, the calling thread blocks until all relevant data have definitely been flushed to the disk before it responds to the Proxy about the handling of the request. The situation is even worse considering that the `fsync(2)` call has to take place *twice* per `Create` request that is handled. Therefore, the latency of the disk I/O is totally shouldered by the calling thread and it is added to the time that is required to handle the incoming `Create` request from the Proxy – in fact, it constitutes the greatest portion of it. As a result, as long as this storage engine is used, the performance of both `MICAS` and `RICAS` is heavily impaired, perhaps to the point that any sort of benchmarking is futile, and possibly even disorientating as far as the actual purpose of this thesis is concerned.

Popular, production-ready storage engine implementations employ mechanisms to mitigate the latency that is inevitably introduced by the use of `fsync(2)`. Most of them rely on some "clever" way to batch multiple modified pages that need to be flushed to the disk before making the actual call to `fsync(2)` – or preferably to `fdatasync(2)` whenever applicable. Until a storage engine of this kind is plugged as the storage backend of our system, both `MICAS`' and `RICAS`' performance is, more or less, doomed to suffer.

4.2 External API

In this section we present the external API of the system. This is the API that is used by clients to communicate with the system in order to store or retrieve data. In both MICAS and RICAS, it is implemented by the stateless frontend tier, the Proxies.

The external API of our systems is designed and implemented in two iterations, both of them over HTTP. The iterations differ from each other in that they leverage principles from different paradigms: the first is RESTful whereas the second is RPC-based. For the time being, due to the simplicity of our requirements, it is hard for any of the two paradigms to preponderate over the other. Nonetheless, the second one turned out to be more efficient in terms of performance, thus it is the only API implementation that is included in the final version of both MICAS and RICAS.

4.2.1 REST, JSON & Base64

First, we will go through the RESTful version of the external API, as it was the first that was developped. It is assumed that the reader is already aware of the arguments presented in section 3.2, and particularly in subsection 3.2.1, about REST, HTTP, JSON and Base64. Below, we reveal the URL endpoints and the associated HTTP verbs that define it:

- `GET /api/v0/objects/<object_ID>`
  Issue a `Read` operation on object with key `<object_ID>`. If the requested object is found, the response's status code is `200 OK` and the Base64 representation of the blob is included, JSON-serialized, in its body. If the requested object is not found, the response's status code is `404 NOT FOUND` and its body is empty. If all Proxy's requests to the Agents that are responsible for the object with key `<object_ID>` time out, then the HTTP status code of the response is `504 GATEWAY TIMEOUT` and its body is empty. On any other case of failure, the response's status is `500 INTERNAL SERVER ERROR` and its body is empty.
- `POST /api/v0/objects/`
  Issue a `Create` operation. The request's body must contain the Base64 representation of the blob to be stored. On success, the HTTP status code of the response

is `201 CREATED` if a new object has been created, or `200 OK` if the object already existed. The body of the response contains the Base64 representation of the key for that blob, JSON-serialized. If the Agents report failure storing the blob to the Proxy, the HTTP status code of the response is `502 BAD GATEWAY` and its body is empty. If one – or more – of Proxy's requests to the Agents that are responsible for storing the blob time out, then the HTTP status code of the response is `504 GATEWAY TIMEOUT` and its body is empty. On any other case of failure, the HTTP status code of the response is `500 INTERNAL SERVER ERROR` and its body is empty.

- `GET /api/v0/objects/<range_start>/<range_end>`
  Issue a `List-Range` operation to retrieve a list of all stored objects with keys in the range [`range_start`, `range_end`] of the keyspace. The URL parameters `<range_start>` and `<range_end>` must be values in their hexadecimal representation, as objects' keys are. The response is sent using `HTTP/1.1`'s **chunked transfer coding**[99], a streaming data transfer mechanism where the data stream is divided into a series of non-overlapping "chunks". Every chunk contains a Base64 encoded JSON list of JSON objects, each reporting the presence of an object along with the number of its replicas stored in the system, for all objects with keys lying between `<range_start>` and `<range_end>` in alphanumerical order.

  In the case described above, where the operation is successful, the HTTP status code of the response is `200 OK`. In the case of an error reported by an upstream server (an Agent), the associated chunk is silently skipped. In other words, the `List-Range` operation is implemented in a sort of best-effort way; hence, even in the case of a `200 OK` response, some of the stored objects (even whole chunks of them, actually) may be missing, without any particular notification for the client.

  In general, there are two such types of failures that may cause a chunk of objects to be skipped:

  – Failures that happen *locally*, i.e. at the Proxy that handles the `List-Range` operation.

  – Failures that happen *upstream*, i.e. at one of the upstream servers, the

Agents, communicating with the Proxy that handles the `List-Range` operation. There may either be "perceived" by the Proxy, e.g. in the case of a time-out, or they may be reported to them directly by the Agents through their internal API; e.g. the Proxy may receive a `500 INTERNAL SERVER ERROR` from an Agent for any kind of failure local to that Agent, or a `503 SERVICE UNAVAILABLE` if that Agent has not completed its initial data synchronization yet, and thus could be in an inconsistent state regarding the objects that are actually stored.

- `GET /api/v0/objects/all`
  Issue a `List` operation to retrieve a list every object stored in the system. This is, actually, treated merely as a special case of the `List-Range` operation, where `<range_start>` is the first legitimate object key in the keyspace and `<range_end>` is the last one.

4.2.2 RPC, gRPC & Protocol Buffers

On the next development iteration, the RESTful external API was replaced by an RPC-based API. It is assumed that the reader is already familiar with the arguments presented in section 3.2 and subsection 3.2.2 about communications and RPC in general, as well as gRPC and protocol buffers in particular.

Below, we present the definition of the external API, as implemented by the `RICAS` Proxies (`MICAS` Proxies' is pretty similar), in the specialized protocol buffers' IDL.

```
1  syntax = "proto3";
2
3  package mpb;
4
5  Service Proxy {
6      rpc Create(ProxyCreateRequest) returns (ProxyCreateResponse) {}
7      rpc Read(ProxyReadRequest) returns (ProxyReadResponse) {}
8  }
9
10 message ProxyCreateRequest {
```

```
11      bytes blob = 1;
12      int32 consistency_level = 2;
13  }
14
15  message ProxyCreateResponse {
16      bytes digest = 1;
17  }
18
19  message ProxyReadRequest {
20      bytes digest = 1;
21  }
22
23  message ProxyReadResponse {
24      bytes blob = 1;
25  }
```

Listing 4.5: *Definition of the Proxies' external gRPC API in the proto3 IDL.*

It is obvious that, again, the API consists of the bare minimum operations required to be functional according to the needs and prerequisites that were described in the previous chapters. The functionality of each message and its fields are self-evident as well.

The `Proxy Service` consists of the two basic RPC operations, `Create` and `Read`, which obviously correspond to the `Create` and `Read` requests, respectively. Each of these remote procedure calls includes its arguments and return values in the form of `messages`:

- The `ProxyCreateRequest` is used when issuing a `Create` remote procedure call. It includes the parameters of the `Create` request, namely the "*adjustable consistency level*" (which has been described thoroughly in chapter 3) for the blob, as well as the actual data of the blob.
- The `ProxyCreateResponse` is returned by the Proxy as a response to a `Create` remote procedure call. It contains the key of the newly created object: its hash digest as produced by the system using the hash function of choice. It can be

later used in `Read` calls to retrieve the stored data of the blob.

- The `ProxyReadRequest` is used when issuing a `Read` remote procedure call. It includes the key of the object to be retrieved, as returned by a previous `Create` operation in the corresponding `ProxyCreateResponse`.
- The `ProxyReadResponse` is return by the Proxy as a response to a `Read` remote procedure call. It contains the data of the requested blob, it is indeed stored in the system.

In any case of error, both RPC operations return a gRPC error value instead of a normal response, which may contain detailed information about the failure that occurred.

We note again that the `List` and `List-Range` operations have been deprecated. They are not included at all in the final versions of the systems, thus neither do they in the external gRPC API.

4.3 Hashing

In this section we pick up where we left off in Section 3.5, where hashing and a relevant part of our system's design and requirements were examined. We argued that in order to ensure collision resistance as required by content-addressability, we need to use a uniform hash function, which produces digests of particular size, at least 128 bits and preferably not bigger than 256 bits, due to the tradeoff between storage space and the probability of collision.

We also noted that cryptographic hash functions have properties that usually make them good candidates for satisfying our requirements. In fact, some of the content-addressable systems that have been previously mentioned in the thesis, like Git[110], Venti[157] and CASPER[158], use the SHA-1 cryptographic hash function [164] developed by the US National Institute for Standards and Technology (NIST) in 1995. It produces 160-bit digests, thus the probability of collision between any two digests is less than 10^{-20}, even for a very large number of stored objects in the order of $\sim 2^{48} \approx 2.8 \times 10^{14}$. SHA-1 is indeed a decent choice, but only if one disregards the security concerns that were raised in Section 3.5, since it was recently proven to be

practically vulnerable [165]. Conveniently, good implementations of the SHA-1 hash function exist in both Python's and Go's standard libraries: Python's includes it in module `hashlib`, and Go's includes it in package `crypto/sha1`.

At the lowest values of the acceptable range of digest sizes, the 128-bit MD5 cryptographic hash function [166] is widely used, albeit severely insecure [167] – security-wise, it is a worse choice than SHA-1. If the expected maximum number of objects stored in the system allows us to use 128-bit hash digests while keeping the probability of collisions low, newer hash functions exist, which are usually computationally more efficient as well. Examples include MurmurHash[168], and Google's CityHash[169], FarmHash[170] and HighwayHash[171, 172], which are all non-cryptographic hash functions. Indeed, the latter presents security claims whereas the former three are considered cryptographically vulnerable [173]; yet their digest size and uniformity might still render them collision resistant enough for certain use cases.
A very fast, native Go implementation of HighwayHash was open sourced by Minio, Inc. not long ago [174, 175], which can make use of Intel and AMD CPUs' `AVX2` instruction set for increased performance benefits.

Initially, our default choice was SHA-256, a cryptographic hash function that is part of SHA-2, which was developed by NIST [164] in 2001. The 256-bit hash digests that it produces, along with its strong cryptographic properties, assures us that its use for content-addressable storage applications is both secure and safe in terms of our analysis in Section 3.5. Being widely used across many kinds of applications, it has been very easy to find good, open source implementations of it in both Python and Go: it exists in both languages' standard libraries, in Python's `hashlib` module and in Go's `crypto/sha256` package.

During another iteration of the implementation, later, and after a relevant benchmarking process, SHA-256 was replaced by BLAKE2b[176] as our default choice of a hash function. BLAKE2 is a cryptographic hash function developed in 2012 as an improvement of the SHA-3 finalist BLAKE. BLAKE2b is one of its flavors, optimized for 64-bit platforms. It produces digests of any size between 1 and 64 bytes, but we stick by our choice of 256-bit (32-byte) ones. Go Sub-repository Packages[7] include a very fast implementation of BLAKE2b, which is also capable of taking advantage of the `AVX2`

[7] Packages that are part of the Go Project but outside the main Go tree (`https://godoc.org/-/subrepo`). They are developed under looser compatibility requirements than the Go core.

instruction set of the CPU for improved performance. The name of this package is `golang.org/x/crypto/blake2b`.

4.4 Consistent Hashing Ring Data Structure

As we have already mentioned in chapter 3, `RICAS`, which employs a consistent hashing algorithm, uses an in-memory data structure to represent the consistent hashing ring, and to calculate all required information that is related to it. Operations on the data structure may be either read or write: read operations include simple retrieval of stored information, whereas write operations modify the stored state of the ring.

But first, let us remember which components of `RICAS` in particular need to have access and keep a consistent hashing ring data structure up to date. The Proxies are the most obvious – and, indeed, the main – candidates for this. Since client requests arrive on the frontend tier first, it is the Proxies that need to figure out which `RICAS` Agents should each object be assigned to, after its hash digest (i.e. object key) is calculated. However, other than the Proxies, the Agents themselves also need to be aware of the topology of the consistent hashing ring under certain circumstances. For instance, when an initial data synchronization or a scale-in data synchronization takes place, every Agent needs to know which other Agents are responsible for storing replicas of the objects in some of the virtual nodes on the ring.

There is a major difference between the Proxies and the Agents concerning the manner they handle their respective consistent hashing ring data structures, though. By both the design and the implementation of the Agents, when an initial data synchronization or a scale-in pull has to occur, the consistent hashing ring data structure remains unmodified for the whole duration of the procedure. For example, a failed-over Agent that begins its initial data synchronization, bases it on the state of the consistent hashing ring at the time it is restarted. If this state changes during the initial data synchronization, e.g. if a scale-in operation is issued cluster-wide, the data structure, held and read by the `Syncer` component-thread of the `RICAS` Agent, remains unmodified; it only changes *after* the initial data synchronization is finished, in order to reflect the new topology and subsequently perform the pending scale-in pull.

Contrary to that, the Proxies use the consistent hashing ring data structure perpetually.

Every single incoming `Create` or `Read` client request signifies a read operation on the data structure. Such requests may arrive all the time, including when the consistent hashing ring is scaled out or in and thus its topology changes. In the latter cases, both read and write operations on the data structure must be handled concurrently, while preserving the soundness of the related functionality.

In the first development iteration, the data structure was developed as a class in its own separate Python module, which was then imported by both the Agents and the Proxies, and it merely included the required functionality for our case. The aim of this class has been to explore the technical requirements of such a data structure, and to provide as much functionality as possible while keeping the implementation simple. Of course, the supported operations on the data structure should be "concurrency-enabled", since multiple threads on the Proxies (the threads that handle the incoming client requests) might need to read and write it concurrently, as explained above; hence, a simple coarse-grained locking mechanism was employed for the coordination among the threads.

In the second development iteration, it is developed as a separate Go package, again imported by both the Agents and the Proxies. However, this time extra care has been paid to the concurrency handling mechanism, which is now based on the RCU[8] synchronization mechanism [194]. The general pattern, which has been applied, is that an update to the data structure can be prepared by reading its state and making a copy of it. Later, the update is installed by leveraging some machine primitive to atomically change just a single pointer.

As pointed out by [194], referring to the RCU implementation in the Linux kernel: "*RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete*".

Indeed, in our case, the data structure is immutable and lives in a memory location

[8]Read-Copy-Update.

shared by multiple threads. Multiple readers have access to it concurrently, whereas only a single thread is responsible for updating it whenever it is required. When the writer updates the state, multiple concurrent readers retain their access to a copy of the previous state of the consistent hashing ring data structure for as long as they need it. When no more readers access it anymore, that copy is garbage-collected by the Go runtime. According to the definitions of the *progress conditions* that a concurrent algorithm or data structure may provide (e.g. [195, 196]), all read and write operations of our implementation are **wait-free**; hence, so is the whole consistent hashing ring data structure. As a matter of a fact, since its performance does not really depend on the number of active threads in any way at all, it actually is **wait-free population-oblivious (WFPO)** [9].

The consistent hashing ring data structure is implemented as the separate package `github.com/ckatsak/lfchring`. It is released separately from the rest of the `RICAS` and is hosted by GitHub; it can be found at `https://github.com/ckatsak/lfchring`. Its documentation is available online at `https://godoc.org/github.com/ckatsak/lfchring`.

4.5 Data Synchronization

It has been repeatedly stated that, currently, all kinds of data synchronization procedures (i.e. both the initial data synchronization performed during failovers and scale-out operations, and the scale-in pull performed during scale-in operations) are based on rsync. In this section, we are examining, in a little more detail, how rsync is used by `MICAS` and `RICAS` to achieve their purpose.

First of all, in both `MICAS` and `RICAS`, any data synchronization required is initiated by a separate component-thread: the `Syncer`. Of course, since the functionality of each of the two systems differs, the exact job of `MICAS`' and `RICAS`' `Syncers` varies as well. So does the architecture at the code level. Nevertheless, there is no data synchronization happening outside of the context of a `Syncer`, which may possibly make a future upgrade a little easier, by implementing a different logic for the `Syncer` and plugging

[9] Of course, in practice, the progress condition may be bounded by other mechanisms, such as the memory allocation mechanism [196], which is indeed put into use in our case.

it in its place. Anyway, as it should have been clear by section 4.1, the logic of the `Syncer` (together with that of the `GarbageCollector`) depends largely on the storage engine that is being used.

4.5.1 MICAS

Let us start by `MICAS`, where the situation is quite simple. We recall that the initial data synchronization of a `MICAS` Agent happens by contacting all other Agents in the same `StatefulSet` as itself, because all Agents within the same `StatefulSet` correspond to a single shard, thus store exactly the same data. Furthermore, since the number of shards (and `StatefulSets`) can only be statically defined at the time of the deployment of `MICAS`, data stored by the system cannot be resharded, so the object keys that are assigned to the Agents of each `StatefulSet` do not change; scaling the system out and in can only occur by rescheduling the existing `StatefulSets` properly.

As shown in Figure 3.5, a booting `MICAS` Agent can find itself in one of the Bootstrapping or Syncing phases. If the cluster is not in the Bootstrapping phase, i.e. if it has not just been deployed, the Agent is in the Syncing phase and needs to perform its initial data synchronization. No other kind of data synchronization is supported by `MICAS` at all. In addition, since the objects assigned to each Agent do not change, the initial data synchronization only occurs when the Agent is restarted.

These observations lead to a straightforward implementation. Upon booting, the `MICAS` Agent forks off another thread [10]. Now that there are two threads, one of the threads acts as the `BlobServer`, serving requests incoming from the Proxies, and the other one as the `Syncer`. All that the latter has to do is to figure out whether the Agent should perform the initial data synchronization by checking whether the Bootstrapping flag is set, and then to initiate it, if it is needed indeed. When this is over, the thread is not needed anymore, thereby it ceases to exist.

Thanks to the design of the sharding and replication in `MICAS`, one single rsync client process per peer Agent in the `StatefulSet` suffices for an Agent to be sure it has retrieved all data that it should own. Therefore, the total number of rsync client processes

[10] Again, technically, it forks off a goroutine. However, the relation between goroutines and operating system threads, which is determined by the Go runtime, will not be discussed here. At the level of this discussion, the reader can simplify the rationale by merely thinking of them as threads.

spawned by each Agent upon being restarted is always $k - 1$. Of course, thanks to the the efficiency of rsync for determining which files should be transferred, usually most of the missing data should be retrieved by the first rsync client process already.

4.5.2 RICAS

The case of `RICAS` is more complex than that of `MICAS`, for a number of reasons. First, the procedure of the initial data synchronization of a `RICAS` Agent is carefully designed and implemented to behave correctly in the case of both a fail-over and a scale-out operation. Although the handlings of these cases of the consistent hashing ring have many similarities, they still are different, and the exact behavior of the Agent depends on the the state of the `RICAS` cluster as a whole, as well as on the role of the particular Agent in each context.

Secondly, other than the initial data synchronization, `RICAS` Agents also support scale-in data synchronization, which is generally pretty unlike the former in many aspects. One of the most important ones is that, contrary to `MICAS`, a scale-in data synchronization may have to be initiated any time during an Agent's lifetime, not just in its beginning. For that reason, unlike in the case of `MICAS`, the corresponding `Syncer` thread in a `RICAS` Agent may not simply cease to exist after the initial data synchronization is over. Instead, it is one of the always-running threads that comprise a `RICAS` Agent, much like the `BlobServer` is (which, by the way, has the same role as in `MICAS`: to listen for and serve requests incoming from the Proxies).

Thirdly, `RICAS`' support for scaling the number of shards in the system up and down means that the object keys that are assigned to each shard change over time. As a result, a single rsync client process may not suffice to finely pick the objects in the appropriate ranges of the keyspace (the ring segments that correspond to the virtual nodes), and only them, efficiently, especially since these ranges can be different from time to time depending on the state of the cluster.

For these reasons, the `Syncer` in `RICAS` is a goroutine running an infinite loop, receiving events to determine its actions. This is achieved via a buffered channel[11], which

[11] A built-in construct in Go for inter-goroutine communication, which is not further discussed here, though.

effectively works as a blocking event queue in this case. The events are always sent by the `AgentController`, another component-thread of the `RICAS` Agent that is responsible for keeping the Agent up to date regarding the state stored in the `RicasCluster` custom Kubernetes resource. They may signify that an initial data synchronization or a scale-in pull should begin, or that the `Syncer` should gracefully shut down, when the Agent is about to be terminated.

The `Syncer` in `RICAS` is, of course, based on rsync, too, which works at the level of files or directories. In other words, to transfer data using rsync, one must specify the local and possibly also the remote paths of the corresponding files. Perhaps now it is obvious why the mechanism used for the data synchronization procedures depends greatly on the storage engine that is being used. In our case, rsync is a good choice because of our naive storage backend implementation, which is based on the underlying filesystem for both storing and also indexing the data, as it was explained in section 4.1. The exact path of each object that the `Syncer` should transfer depends on the index and its configured level. In the case of `MICAS` the situation is simple: since all Agents in a `StatefulSet` are responsible for the same data, transferring the data using the root of the index is good enough.

However, in the case of `RICAS`, the keyspace is splitted into multiple ring segments matching the plethora of virtual nodes that are present on the consistent hashing ring. On each data synchronization, in order to finely pick only those ring segments that are required to be transferred, multiple rsync client processes must be forked off. Let h_A and h_B denote two N-bit hash digests of the virtual nodes S_{A_x} and S_{B_y} (of Agents S_A and S_B, respectively) that lie next to each other on the consistent hashing ring, and $h_A < h_B$. Assume that a failed-over Agent has to pull the data of S_{B_y} as part of its initial data synchronization. The objects that are assigned to it, in the current state of the consistent hashing ring, are those whose keys lie in the range $[h_A + 1,\ h_B]$ [12].

At first glance, to transfer the data of the virtual node, the Agent could fork off one rsync client process per index-leaf subdirectory that contains one or more objects in the range $[h_A + 1,\ h_B]$ – excluding the subdirectories at the two ends, for now, to avoid transferring objects with keys outside this range. For example, let us assume the values $h_A =$ `0x4b2d4ce23d88e2ee9568b` and $h_B =$ `0x6abf281df5a5d0ff3cad6`

[12] Without loss of generality, we can assume that $0 \leq h_A < h_B$, to simplify the current discussion.

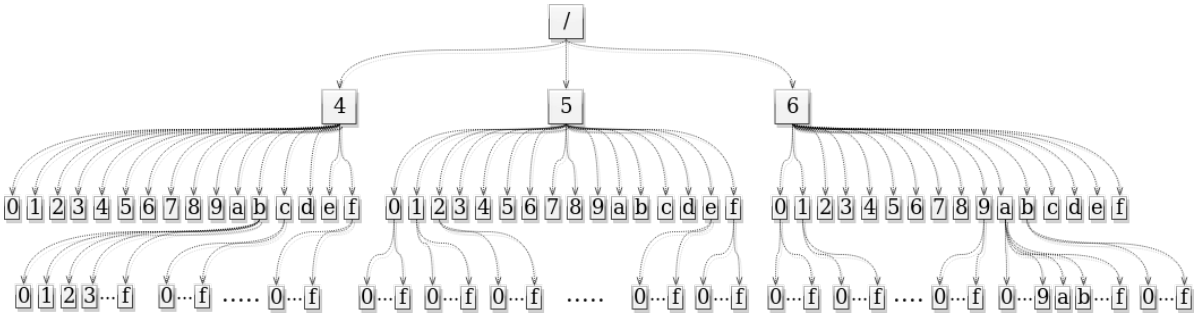


Figure 4.3: *A part of an index tree example, where the index height is set to three. Not all subdirectories are shown in this figure, for practical reasons; only the part of "special interest" in the context of our example is shown, in a compact form.*

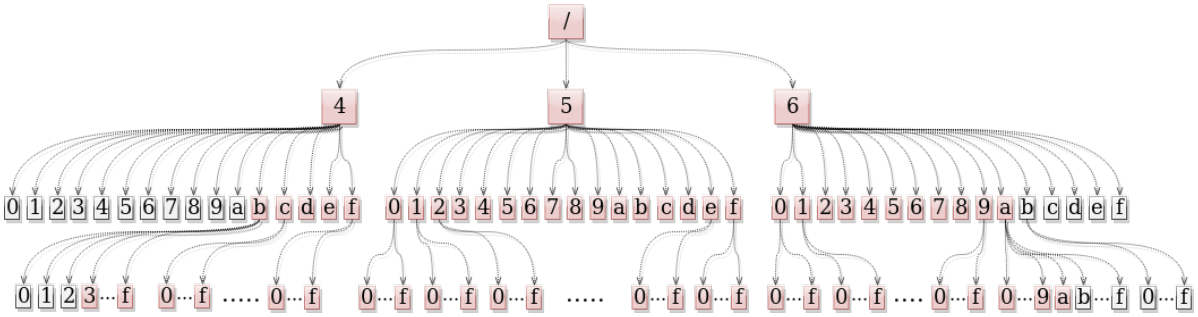


Figure 4.4: *The same example as in Figure 4.3, but this time the subdirectories that should be transferred "as is" have been additionally marked with color.*

and an index level of three where the index is rooted at `$INDEX_LEVEL/`. Figure 4.3 depicts a part of the index tree (which has special interest in the context of this example), and in Figure 4.4 the subdirectories that should be transferred during this data synchronization have been additionally marked. The Agent could attempt to pull *every* index-leaf subdirectory under this root in the range: `4/b/3/`, `4/b/4/`, …, `4/b/f/`, `4/c/0/`, …, `4/f/f/`, `5/0/0/`, …, `6/a/a/`, one by one, traversing the index tree as shown with the blue arrow in Figure 4.5. However, due to the relatively large fanout of sixteen, the number of such index-leaf subdirectories can easily become large, as dictated by (4.1). The higher the index level, the greater the overhead caused by spawning a different rsync client process to deal with the transfer of each index-leaf subdirectory.

To reduce the total number of rsync client processes that need to be forked off, we implement an alternative way to traverse the index tree. This way guarantees that the number of directories that are visited during the traverse of the tree is minimum, by finding as many common ancestor-directories as possible in the tree. For instance, in the example presented above, instead of visiting all the index-leaf subdirectories, it suffices to transfer subdirectories: `4/b/3/`, `4/b/4/`, …, `4/b/f/`, `4/c/`, `4/d/`,

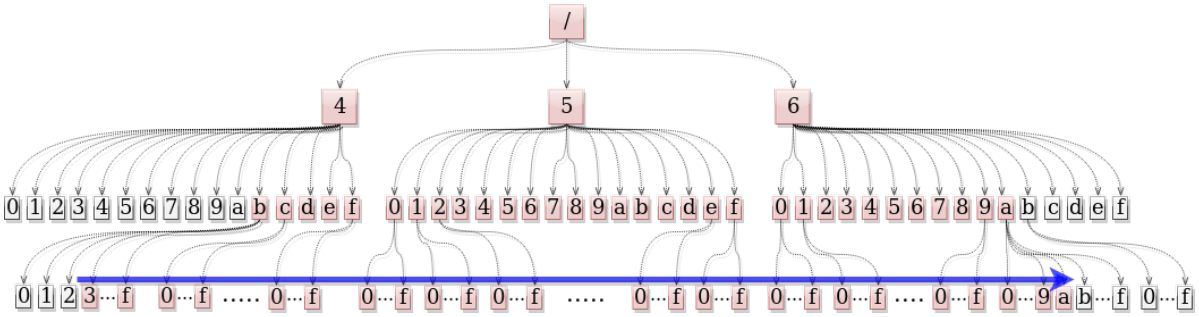


Figure 4.5: *The example of Figure 4.3 and Figure 4.4, with an additional blue arrow indicating the traversal of every index-leaf subdirectory in the index's directory hierarchy.*

`4/e/`, `4/f/`, `5/`, `6/0/`, `6/1/`, …, `6/9/`, `6/a/0/`, `6/a/1/`, …, `6/a/a/`. The index tree is traversed as shown in Figure 4.6, and considering that one rsync process is forked off for each subdirectory, the total number of spawned processes is reduced significantly in this way.

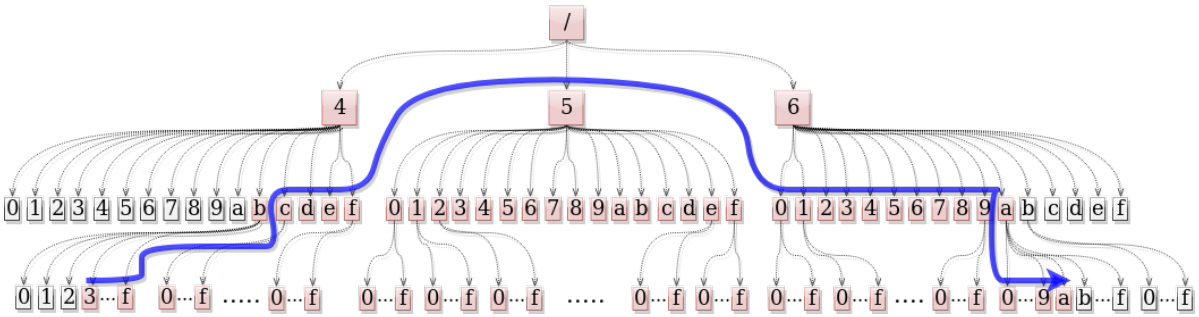


Figure 4.6: *The example of Figure 4.3 and Figure 4.4, with an additional blue arrow indicating the traversal of the index's directory hierarchy that is optimal in terms of the number of rsync processes that are required to be spawned.*

What about the index-leaf subdirectories at the edge of this range, though, which are left out in the procedure above? What is special about them (`4/b/2/` and `6/a/b/` in the previous example) is that they may also contain objects that lie within the ring segment of a different virtual node than the one that has to be pulled by the Agent; namely, of the previous or the next virtual node on the consistent hashing ring. For these objects to be transferred, they need to be picked even more finely than the selection based on index-leaf subdirectories. To achieve that, we use rsync, again, this time enabling its file listing functionality: by specifying the `--list-only` option, an rsync client can be used to list all files stored in a remote directory. Knowing – more or less – the index directory hierarchy on the remote Agent, the puller Agent is able to retrieve a list of the files present in the index-leaf subdirectory at the edge of the range. Next, the Agent

cherry-picks the objects of its interest – the ones that belong to the virtual node it needs to synchronize – and the sorts them into a list. This list of files is then dumped into a temporary file, and the objects that it contains are transferred from the remote host, again using rsync with its `--files-from` option enabled (among others).

This way, all objects under a virtual node's ring segment are transferred, and only these. An alternative solution would be to just transfer every single file under these index-leaf subdirectories, and simply ignore any extra objects that are going to be pulled as well. These objects would not really be accessible by the Proxies, since their corresponding replica owners, based on the consistent hashing ring, do not include this Agent. Therefore, they would just taking up precious storage space, being completely useless; hence, they may even could be garbage collected. However, since they can be numerous, and as large as a few MBs each, this approach would still be a waste as far as the network bandwidth is concerned.

It should be even more evident by now how the data synchronization mechanism of `RICAS` is largely dependent on the underlying storage engine. Had the storage engine been a different one, none of the above would be relevant at all. This is the reason that the `syncer.backend` abstraction exists (as a Go interface, presented in subsection 4.1.1) and needs to be upgraded as well in the case of an upgrade of the corresponding `storage.Backend`, the abstraction for the storage engine that is being used. It is noted that the `syncer.backend` abstraction is only responsible for the actual transfer of the data. The rest of the operations that are required to be accomplished, mostly related with the `RicasCluster` custom Kubernetes API resource, are done by the `Syncer` itself, making it even easier to develop a specialized data synchronization backend when another storage engine should be plugged in.

In addition to the above, our rsync-based `syncer.backend` implementation, called `Rsyncer`, is multi-threaded in `RICAS`. It uses multiple goroutines to spawn many rsync client processes concurrently, so that even if one rsync client's TCP connection does not saturate the link for any reason, e.g. because of some connectivity issue, concurrent rsync clients step in for efficiency. The number of concurrent goroutines of the `Rsyncer` is configurable at the time of the deployment of `RICAS`. Of course, these concurrent goroutines attempt to synchronize the objects of different virtual nodes. Each of them picks its next virtual node by traversing the consistent hashing ring data struc-

ture. If the synchronization of a virtual node fails, it is re-enqueued in order to be retried later, either by the same or by a different goroutine.

To further reduce the number of spawned processes, in both `MICAS` and `RICAS`, the rsync client processes are forked off using the `os/exec` package of the standard library of Go. Instead of using the `fork(2)` system call, rsync client processes are spawned using the `vfork(2)` system call. Pretty much like `fork(2)`, `vfork(2)` is a special case of the `clone(2)` system call. "*It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an* `execve(2)`. *It differs from* `fork(2)` *in that the calling thread is suspended until the child terminates (either normally, by calling* `_exit(2)`*, or abnormally, after delivery of a fatal signal), or it makes a call to* `execve(2)`. *Until that point, the child shares all memory with its parent, including the stack*" [197].

4.5.3 EndpointsWatcher

Except from the – sometimes slight – computational overhead imposed by spawning a large number of rsync client processes, there can be another, more subtle overhead. As we have mentioned before, Kubernetes provides a unique DNS name for every Agent Pod in a `StatefulSet` through KubeDNS. This makes communication with Agent Pods easy, relieving the application from the burden of always knowing the IP address of every Agent Pod it needs to contact. An rsync client is fully capable of resolving a DNS name when given one to perform a file transfer from or to a remote host, and this is what happens in the case of `MICAS`.

However, the situation is different for `RICAS`. As we explained above, in `RICAS`, there is a large number of rsync client processes that are required to be forked off in order to finely pick the objects that must be transferred. In fact, the number is large enough to cause a considerable overhead due to the roundtrip time of the DNS resolution procedure, which involves the Agent and KubeDNS Pods, as well as etcd, where KubeDNS stores the relevant information required [13].

[13] In fact, this information is stored in etcd by SkyDNS[198], which, in turn, is used by the KubeDNS addon.

To deal with this overhead, we decided that DNS resolution should be avoided altogether in the case of `RICAS`. To this end, another component-thread is introduced for `RICAS` Agents: the `EndpointsWatcher`. Its job is simple; it leverages the `watch` operation provided by the Kubernetes API server using Kubernetes client-go library's framework of `Informers` and `Indexers`, in order to keep a locally stored cache always up to date. This cache simply contains a mapping of Agent Pods' DNS names to their respective Kubernetes cluster-internal IP addresses. It is consulted by the Agent's `Syncer` during any data synchronization procedure, right before a new rsync client process is forked off, thus avoiding the need for rsync to resolve the DNS name of the remote Agent.

4.5.4 Rsync Options

Last, we mention some of the command-line options enabled in the rsync client processes that are spawned during every data synchronization:

- `--itemize-changes`
  With this option, rsync produces a compact – but kind of cryptic – output that contains a simple itemized list of the changes that are being made to each file, including attribute changes.

- `--size-only`
  This modifies rsync's "quick check" algorithm for determining which files need to be transferred, changing it from the default of transferring files with either a changed size or a changed time of the most recent modification, to just looking for files that have changed in size. In our case, this option accelerates the data transfer overall, thus is useful, because, thanks to immutability and the atomicity in objects' creation, the time of the last modification is actually irrelevant in deciding which objects have to be pulled. Furthermore, content-addressability by the means of a collision-resistant hash function (as presented in section 3.5) make the odds against a hash collision of blobs of the same size, where this size can only get as large as a few MBs, pretty much astronomical. On the other hand, disabling rsync's "quick check" algorithm altogether (by enabling the `--ignore-times` option) would cause all objects of the remote Agent to be read,

instead of reading merely their filesystem attributes, which would be inefficient.

- `--compress`
  Enabling this option, allows rsync to compress the data in the files as they are sent to the destination, which reduces the amount of data being transmitted.

- `--recursive`
  This tells rsync to copy directories recursively, thus it is useful in both `MICAS`, where the whole index's root directory is given as an argument to rsync, and in `RICAS`, where the optimal index traversal that was described previously in this section may use a non-index-leaf subdirectory as rsync client's argument.

- `--ignore-missing-args`
  When rsync is first processing the explicitly requested source files and directories (through either the command-line arguments or `--files-from` entries), it is normally an error if the file or directory cannot be found. This option suppresses that error, and does not try to transfer the file. It is useful in `RICAS`, where index's subdirectories need to be traversed and transferred one-by-one, in cases that some of them have not been created yet by the remote Agent (since they are created lazily, as explained in subsection 4.1.3), or have been removed after having been emptied as a result of some garbage collection procedure.

- `--contimeout`
  This option allows setting the amount of time that rsync waits for its connection to an rsync daemon to succeed. If the timeout is reached, rsync exits with an error. Using this option, in conjunction with `--timeout`, we avoid having the whole system hang because of a single rsync client process. Its value can be configured during the deployment of `MICAS` and `RICAS`.

- `--timeout`
  This option allows setting a maximum I/O timeout in seconds. If no data is transferred for the specified time then rsync exits. Using this option, in conjunction with `--contimeout`, we avoid having the whole system hang because of a single rsync client process. Its value can be configured during the deployment of `MICAS` and `RICAS`.

4.6 Garbage Collection

Garbage collection of stored objects is required whenever the set of object keys that are assigned to an Agent changes during the operation of the system. As we have already seen, this can happen as a result of a scale-out or scale-in operation in general, but only in the case of a scale-out operation in the case of `RICAS`, in which new virtual nodes are added to the consistent hashing ring. However, it cannot happen at all in `MICAS`, where the number of shards is static, thus the objects assigned to each shard is static, too.

The `GarbageCollector` is a component-thread of every `RICAS` Agent responsible for handling the local garbage collection of stored objects and communicating its status via the `RicasCluster` custom Kubernetes API resource (details about these have been presented in subsection 3.9.4) and participating in the coordination of the Garbage Collection phase across the whole cluster. The implementation is very similar to that of `RICAS`' `Syncer`: the `GarbageCollector` is essentially a goroutine running in an infinite loop, receiving events to determine its actions. The events are delivered via a buffered channel that works as a blocking event queue, and they are sent by the `AgentController` component-thread when the cluster enters the Garbage Collection phase.

Similar to the `Syncer`, the implementation of the `GarbageCollector` is largely dependent on the underlying storage engine that is being used. Unless the exact way that the objects are being stored is known, it is impossible to figure out how they should be removed. This is the reason why the `storage.gcBackend` abstraction exists, as a Go interface, which was mentioned in subsection 4.1.1: upgrading `RICAS` to use a real storage engine instead of our naive one, would require not only to upgrade the `Syncer`, as we saw in the previous subsection, but also the `GarbageCollector`, by satisfying this abstraction. It is noted that the `storage.gcBackend` abstraction is only responsible for the actual removal of the data. The rest of the operations that are required to be accomplished, mostly related with the `RicasCluster` custom Kubernetes API resource, are done by the `GarbageCollector` itself, making it even easier to develop a specialized garbage collection backend when another storage engine should be plugged in.

The implementation of the `gcBackend` interface for our naive storage engine is straight-

forward. The index tree is traversed in a similar way as in the case of `Syncer`, only this time, instead of transferring the directories, they are being removed using the `RemoveAll` function of the `os` package in Go's standard library. This is essentially equivalent to traversing all the index-leaf subdirectories (except from the ones at the two ends) and using the `unlink(2)` system call for every file they contain, and also the `rmdir(2)` system call for each directory, and then gradually moving upwards the directory hierarchy to deal with the parent-directories too. Next, the index-leaf subdirectories at the two ends are visited separately to finely pick only those files that correspond to objects of the virtual node in question only, before using `unlink(2)` for them as well, and possibly `rmdir(2)` for the subdirectories themselves, if they end up empty after this procedure.

4.7 Summary of RICAS Implementation

Last, we present simplified versions of class diagrams for `RICAS` Agents and Proxies. Note that Go is not a standard object-oriented language, like Java and Python are; it does not even support classes to start with. However, it supports and encourages development that leverages many concepts of the object-oriented programming paradigm. Both diagrams are quite simplified, in the sense that not all such "classes" are presented here, and none of their "attributes" and methods are shown here neither. In fact, those presented in the diagram are not even all of the essential ones. They all are, however, the most fundamental of those defined in this project (rather than some external library or framework), and they do constitute, to a large extent, the backbone of the `RICAS` Agent and Proxy implementations.

4.7.1 Agent

The diagram for `RICAS` Agent is presented in Figure 4.7. It is noted that, since almost every cardinality value in the associations (including the aggregations and the compositions) is 1, it is omitted from the diagram to achieve a visually more "clear" result; when the value of such a cardinality is different than 1 it is not omitted. It is also noted that `syncRq` and `gcRq` are not really "classes" (or even custom Go structs, as the rest). They are the buffered channels that are shared between the `AgentController`, which

is the sender, and the `Syncer` and `GarbageCollector`, which are the receivers, and they act more like event queues in this case, as we have mentioned again earlier.

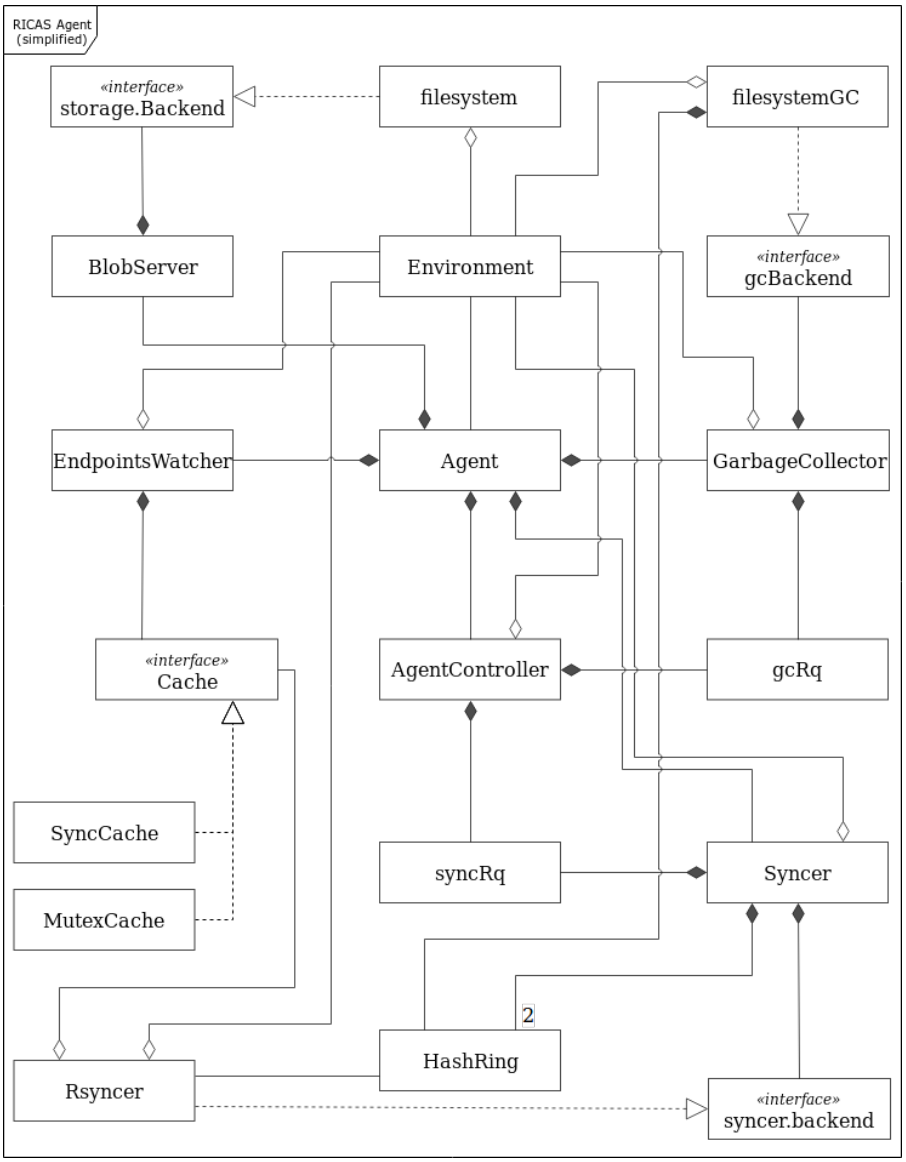


Figure 4.7: *A simplified kind of class diagram for a part of the* `RICAS` *Agent's implementation that consists of "classes" defined within this project.*

Next, we briefly comment on the various "classes" of the diagram:

- `Environment`

  A struct that contains various general static information about the `RICAS` deployment, including deployment-time configuration, such as the types of the storage, syncer and garbage collection backends, the height of the index, the number of virtual nodes per consistent hashing ring node, various timeout val-

ues, etc. Most of them usually are useful to both the Agents and the Proxies.

- `BlobServer`
  One of the major component-threads that comprise a `RICAS` Agent process, which is responsible for serving requests incoming from the Proxies, exposing the gRPC network interface and using the storage backend.

- `storage.Backend`
  A Go interface representing the storage backend abstraction, introduced in subsection 4.1.1.

- `filesystem`
  The naive implementation of our storage engine, which has been described in detail in section 4.1.

- `EndpointsWatcher`
  One of the component-threads that comprise a `RICAS` Agent process, which is responsible for keeping a local cache mapping Agent names to their Kubernetes cluster-internal IP addresses up to date. Its functionality been described in subsection 4.5.3.

- `Cache, SyncCache, MutexCache`
  The interface, and two different implementations of the cache that maps Agent names to their Kubernetes cluster-internal IP addresses. It is local to the `RICAS` Agent and shared by multiple component-threads that comprise it. Its functionality been described in subsection 4.5.3, too.

- `HashRing`
  The consistent hashing ring data structure, which has been presented in more detail in section 4.4, defined in the `github.com/ckatsak/lfchring` package.

- `AgentController`
  Another major component-thread of every `RICAS` Agent. Mainly, it is responsible for watching, and sometimes modifying, the state of the `RicasCluster` custom Kubernetes API object, and for initiating any local operation required so that the Agent acts in accordance with the rest of the `RICAS` cluster. For instance, it is responsible for initiating the local garbage collection, as well as any

kind of data synchronization, upon any such indication by the upstream `RicasCluster` object.

- `GarbageCollector`
  Another component-thread of the `RICAS` Agent, which is responsible for performing the local garbage collection of stored objects when the `RICAS` cluster enters the Garbage Collection phase after a scale-out operation has been completed. It also updates the `RicasCluster` object when it is done. Details have been presented mainly in subsection 3.9.4, but also in section 4.6.

- `gcRq`
  A buffered Go channel for the `AgentController` to notify the `GarbageCollector` to initiate the local garbage collection procedure.

- `gcBackend`
  A Go interface representing the garbage collection backend abstraction, which has been discussed in subsection 4.1.1 and section 4.6.

- `filesystemGC`
  The implementation of the `gcBackend` designed and implemented for our naive storage engine. As we have already discussed, the implementation of the `gcBackend` greatly depends on the actual storage engine that is being used. More details about its functionality can be found in section 4.6.

- `Syncer`
  Another component-thread of the `RICAS` Agent, which is responsible for performing all kinds of data synchronization, whenever one of them is needed, i.e. on an Agent's first booting, fail-over, or when the cluster is scaled in. It also modifies the `RicasCluster` properly in each of these circumstances. Much of its functionality has been discussed all over section 3.9, and also in section 4.5.

- `syncRq`
  A buffered Go channel for the `AgentController` to notify the `Syncer` to initiate some kind of data synchronization procedure.

- `syncer.Backend`
  A Go interface representing the data synchronization backend abstraction, which

has been mentioned in subsection 4.1.1 and discussed in section 4.5.

- `Rsyncer`
  The rsync-based implementation of the `syncer.Backend` designed and implemented for our naive storage engine. As we have already discussed, the implementation of the `syncer.Backend` greatly depends on the actual storage engine that is being used. Details about its functionality are discussed in section 4.5.

- `Agent`
  An aggregate "class" that "owns" all other component-threads of the `RICAS` Agent. It is manipulated by the main thread and, through it, all other component-threads are properly initialized, spawned and gracefully shut down when the Agent needs to be terminated.

It is noted that the implementation `BlobServer`, `Syncer` and `GarbageCollector`, together with their counterparts, `storage.Backend`, `syncer.backend` and `storage.gcBackend`, is developed based on the "Gang of Four"[199] structural design pattern known as Bridge.

4.7.2 Proxy

The diagram for `RICAS` Proxy is presented in Figure 4.8. Again, every cardinality value that is omitted from the associations in the diagram (including any aggregations and compositions) is 1. Details about the functionality of Proxies were presented all over section 3.9, and especially in subsubsection 3.9.4.7.

Next, we briefly comment on the various "classes" of the diagram:

- `Environment`
  This is exactly the same struct as in `RICAS` Agents: a struct that contains various general static information about the `RICAS` deployment, including deployment-time configuration, such as the types of the storage, syncer and garbage collection backends, the height of the index, the number of virtual nodes per consistent hashing ring node, various timeout values, etc. Most of them usually are useful to both the Agents and the Proxies.

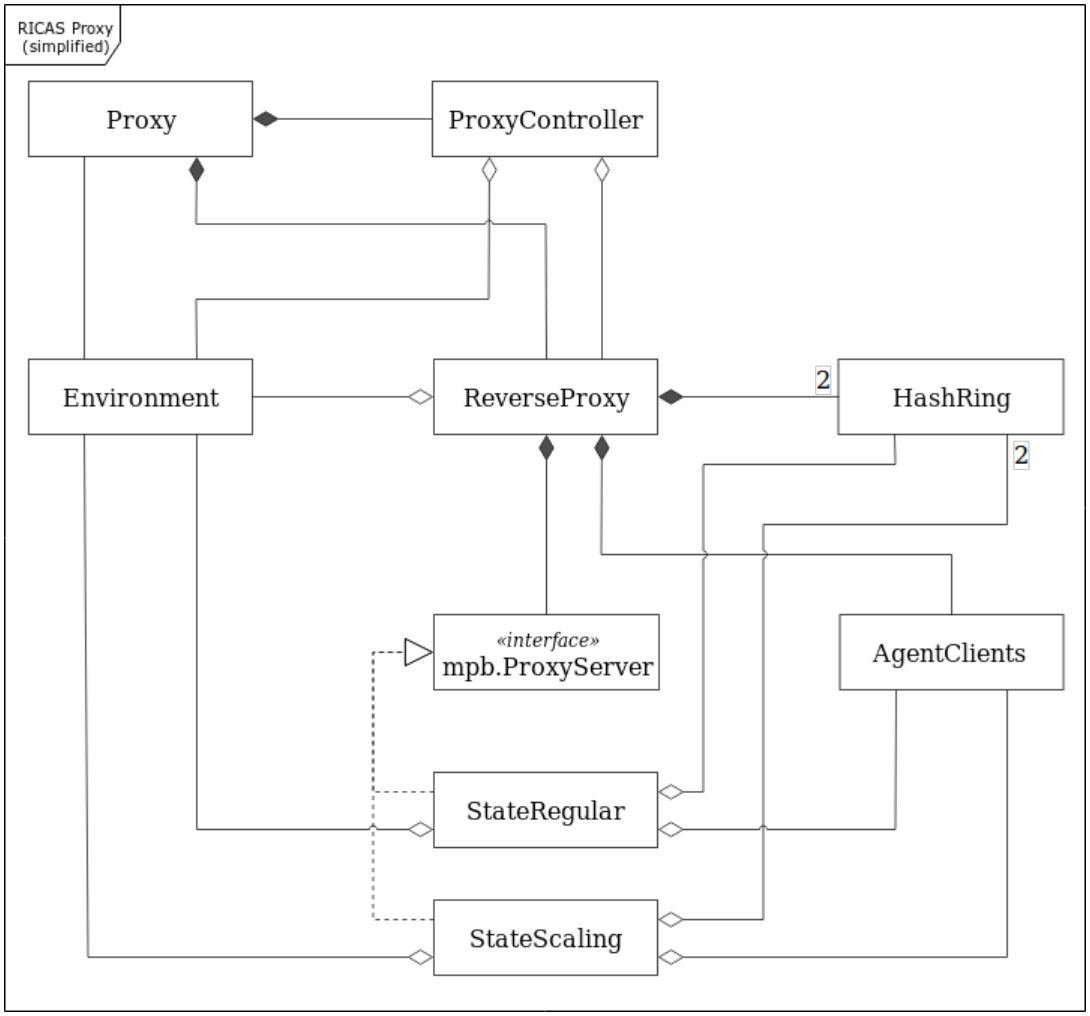


Figure 4.8: *A simplified kind of class diagram for a part of the* `RICAS` *Proxy's implementation that consists of "classes" defined within this project.*

- `HashRing`

  Similar to the case of `RICAS` Agents, this is the consistent hashing ring data structure, which has been presented in more detail in section 4.4, defined in the `github.com/ckatsak/lfchring` package.

- `AgentClients`

  `RICAS` Proxies need to be connected with all the Agents on the backend tier all the time, especially when the load of incoming client requests is heavy. This "class" is responsible for providing a quick and easy access to such gRPC connection handles, decoupling any gRPC connectivity issues and configuration from the rest of the implementation of the Proxy. Note that `AgentClients` itself is merely a concurrent hash map, which, however, consists of other custom data

structures that are not presented here, which, in turn, leverage constructs and functionality of the gRPC library.

- `ReverseProxy`
  One of the component-threads of the `RICAS` Proxy, it is responsible for serving requests incoming from the clients, exposing the external gRPC network API of `RICAS`. It is the "heart" of a `RICAS` Proxy, since it is the one employing the proper sharding and replication technique and orchestrating the communication with the appropriate Agents on the backend tier.

- `mpb.ProxyServer`
  A Go interface, originally defined via the automatic code generation of the gRPC library, representing the abstraction of an entity that is capable of serving gRPC methods of the `RICAS` external gRPC-based API.

- `StateRegular`
  An implementation of the `mpb.ProxyServer` interface that is responsible for handling the client requests when *no* scale-out or scale-in operation is in progress (i.e. most of the time), based on the state of the consistent hashing ring.

- `StateScaling`
  An implementation of the `mpb.ProxyServer` interface that is responsible for handling the client requests for the duration of some scale-out or scale-in operation, based on the state of the consistent hashing ring both before the operation was issued and after the operation is finished.

- `ProxyController`
  Another component-thread of every `RICAS` Proxy. Mainly, it is responsible for watching the state of the `RicasCluster` custom Kubernetes API object, and for initiating any local operation required so that the Proxy acts in soundly when the `RICAS` cluster is in some special transitioning state, i.e. during scale-out and scale-in operations. For instance, it is responsible for atomically switching the active state of the `ReverseProxy`'s request handler between the `StateRegular` and `StateScaling` implementations of its `mpb.ProxyServer`.

- `Proxy`
  An aggregate "class" that "owns" all other component-threads of the `RICAS` Proxy.

It is manipulated by the main thread and, through it, all other component-threads are properly initialized, spawned and gracefully shut down when the Proxy needs to be terminated.

It is noted that the `mpb.ProxyServer` interface owned by the `ReverseProxy` entity, together with its two implementations, `StateRegular` and `StateScaling`, are based on the "Gang of Four"[199] behavioral design pattern known as State. Furthermore, the `ReverseProxy` itself kind of acts as a Mediator (again a "Gang of Four"[199] behavioral design pattern) for the `ProxyController` to switch the above state.

5 Conclusion

In this final chapter we present a brief synopsis of our work, assessing some principal points of the design process. Following that, we conclude by mentioning a few possible extensions and improvements that could be developed in the future, if to make `MICAS` and `RICAS` deployments any meaningful outside experimental purposes.

5.1 Concluding Remarks

Our goal has been to study and experiment with the interoperability of containers and local persistent storage. It is our firm belief that in the modern world of commoditized hardware, huge data sets and cloud-native microservices, containerized distributed storage systems could undeniably have a place in the datacenter. Their development, packaging and, of course, deployment can benefit from the advantages of containers and the convenience from the features provided by orchestration tools, like Kubernetes, and their thriving ecosystem.

We developed two variants of a distributed object storage system. Both of them leverage well known concepts, such as **data immutability** in conjunction with **content-addressability**, **hashing** and **sharding**. Moreover, they both are **highly available** as far as `Read` operations are concerned, and, thanks to the "*adjustable consistency level at per-object granularity*", they can retain high availability for `Create` operations too, although with some caveats. **Fault tolerance** is, of course, of utmost importance for retaining high availability, and it is achieved via **data replication** with a replication

factor that can be configured at the time of the deployment.

Both MICAS and RICAS, the distributed object storage systems that have been developed during this thesis, are **self-healing**. This is a great feature that has been supported pretty effortlessly on our side: the scheduling, status checking and failover of our systems' running components have been delegated to Kubernetes via its API. By carefully elucidating the needs and expectations of the systems, we have been able to choose the Kubernetes API objects that fit better to our deployment requirements. In fact, sometimes, these requirements – and the corresponding features of our systems – were adjusted to or even built on Kubernetes' available functionality and features. However, the "application logic" of the distributed storage systems, such as the sharding and **load balancing**, as well as the data synchronization on failover or scaling operations, are clearly a part of MICAS' and RICAS' logic, of course.

The two systems employ different sharding and replication techniques that yield dissimilar features and qualitative characteristics for each one of them. In the first system, MICAS, sharding and replication is achieved via **mod-N hashing** over a **static number of shards**. The system is **horizontally scalable** in the sense that the number of shards can be statically defined, at the time of the deployment, according to the needs and expectations of each specific use case.

Due to the staticity of the number of shards in MICAS, and the consequent structural rigidity of the system overall, RICAS was developed to use a different sharding and replication mechanism. RICAS implements the battle-tested **consistent hashing** algorithm (with support for **virtual nodes**), which has been employed by several distributed storage systems over the last years, including, but not limited to, Amazon Dynamo, Cassandra and Riak. In addition to the benefits and features that have been explained so far, RICAS provides **elasticity**: the number of shards can be modified dynamically while the system is in use. To this end, RICAS also supports "*control operations*", except from the usual "*data operations*" that are supported in MICAS, which may be issued either explicitly, such as *scale-out* or *scale-in* operations, or implicitly, such as *garbage collection*.

The development of MICAS and RICAS distributed storage systems during this thesis, in spite of their naiveté, makes, if anything, one strong point: Kubernetes, which, despite its young age, is already being used across the industry as a platform for develop-

ing and deploying stateless applications in production environments, is actually a lot more than a simple orchestration tool. Its rich RESTful API comprises of a plethora of options capable of satisfying a variety of deployment requirements. Except from that, however, being backed by etcd and its consistency guarantees, the Kubernetes API server together with the excellent support of its client library for the Go programming language, `client-go`, allows for painless integration of stateful applications and systems with Kubernetes.

Operators and CRDs[1], leveraging the controller pattern through the `informers` and `indexers` framework of `client-go`, which are even used in Kubernetes' codebase itself, play a major part in the process of the design and the implementation of systems to this end. Indeed, it is them in the center of the most recent trend of porting stateful applications and systems to Kubernetes, a procedure that has already many attempts, e.g. for Ceph[32], etcd[200]. Kafka[201, 202], PostgreSQL[203, 204], MySQL[206] and Cassandra[207], many of which are successful. Born in this trend, but also enhancing it, there is the recently developed Operator Framework[208], "*an open source toolkit to manage Kubernetes native applications, called Operators, in an effective, automated, and scalable way*".

5.2 Future Work

Pretty much as expected, both `MICAS` and `RICAS` are far from completed or suitable to be deployed in a production environment in their current state. However, there is a number of ways that the systems could be extended in order to be improved, mostly in terms of indispensable features that such distributed storage systems should have, but they are currently missing.

- **Storage engine**
  As we have discussed previously in chapter 3 and chapter 4, the storage engine that is currently being used in both `MICAS` and `RICAS` has been designed and implemented as a part of this project. However, since focusing on the lower-level storage components of the systems does not fall within the scope of the thesis,

[1] Kubernetes's `CustomResourceDefinition` API resources, through which custom Kubernetes API resources may be defined.

this storage engine essentially is a thin layer over the underlying filesystem's abstractions and it handles the data, especially write operations, in an extremely naive manner. Instead of this, in the future, a proper storage engine should be plugged in to be used as the storage backend of the backend tier of the systems instead. This is expected to boost the overall performance of all data operations of `MICAS` and `RICAS` significantly.

- **Data repair mechanism**
  A feature much needed in `RICAS` is some sort of implicitly invoked "Repair" *control operation*, along with corresponding *Repair* phases for both the cluster and the Agents. As we discussed in section 3.9, the feature of "adjustable consistency level at per-object granularity" effectively allows a number of Agents to entirely ignore the creation of a new object, every time that consistency is relaxed in favor of availability. Due to the workings of the current rsync-based data synchronization process, under certain failure circumstances, the ignorance about such an object may be propagated among the Agents, thus harming the high-availability guarantees of our system. A periodic repair phase, during which no other control operation could be issued, would allow all `RICAS` Agents to consult one another to figure out what objects each one of them may be missing, and retrieve them.

- **Kubernetes' local PersistentVolumes**
  As we discussed in chapter 3, at the time of the design and implementation of `MICAS` and `RICAS`, Kubernetes' options for providing local persistent storage were quite limited. For that reason, we ended up using the simplest type of Kubernetes `Volumes`, the `emptyDir`. Since `Volumes` of the `emptyDir` type are destroyed whenever its associated Pod is destroyed, along with their data, this has slightly affected the design and the implementation of our system. For instance, `MICAS` cannot recover from a failure of all k Agents of a single shard-`StatefulSet`. Similarly, `RICAS` may be unable to recover from the failure of any k Agents across the whole `RICAS` cluster, since, due to the presence of virtual nodes, each shard is actually splitted among multiple Agents. In such cases of failures, it is possible that every Agent that should have a replica of some object are failed, along with their `emptyDir Volume`, so no replica of the object

remains stored in the system at all.

However, thanks to Kubernetes' rapid development cycles, nowadays, local peristent storage is supported via `PeristentVolumes` and `PersistentVolumeClaims`. In this case, the lifetime of the `PersistentVolume` is not tied to the lifetime of the associated Agent Pod. Therefore, in the cases of k failures that are described above, the data synchronization may be delayed, but eventually it could be completed correctly; hence, recovery is possible. Both `MICAS` and `RICAS` should be adjusted to make use of this feature, perhaps also slightly modifying their logic with respect to the handling of such cases of failure.

- **Storage capacity awareness**

  After the previous point has been implemented, perhaps it would be feasible to make storage capacity-aware load balancing of the data among the `RICAS` Agents. Consistent hashing in `RICAS` is basically achieved using the `lfchring` package that we developed. As we have previously mentioned, for now, `lfchring` uses a static number of virtual nodes per node, which is defined at the time of the deployment. This package can be modified to assign a different number of virtual nodes to each Agent according to its underlying storage capacity. Agents that have more space to store objects than others, will be assigned more virtual nodes than them. Effectively, this is a way to adjust the size of the ring segments designated to each Agent on the consistent hashing ring, thus adjusting the size of each shard. In this way, assuming the i-th Agent is designated V_i virtual nodes, the fraction of the total objects assigned to each Agent should be

  $$\frac{V_i}{\sum_{j=0}^{N_s-1} V_j}$$

  where $i \in [0,\ N_s - 1] \subset \mathbb{Z}$, with N_s being the number of deployed `RICAS` Agents.

- **Caching**

  Immutability has been thoroughly discussed in chapter 3. However, there is an important beneficial aspect of it that we have not really touched until now, and it related to caching. In general, caching is used to improve the performance of accessing some resource. When multiple caches are used for the same resource,

cache coherence issues may emerge, i.e. we have to make sure that all caches of the resource have the same data, and that this data is consistent.

In our case, all data are immutable. *Immutable data never expire*. Therefore, they are perfect candidates to populate any number of caches, without actually creating any cache coherence problem. The architecture of both `MICAS` and `RICAS` is multi-tiered: Proxies on the frontend are the first to receive client requests, properly redirecting them to the Agents on the backend that are responsible to store them. Since every single object stored in `MICAS` and `RICAS` is immutable, all data can be cached relentlessly by any number of Proxies. Caching is a powerful technique, especially during heavy workloads that comprise mostly of `Read` requests. Such incoming requests for objects that are queried often, can greatly benefit from finding the objects cached on the frontend, in terms of both latency, as perceived by the client, and network bandwidth utilization within the datacenter.

- **Update operations**
  As we have discussed in chapter 1 and chapter 3, none of the two systems supports Update operations. They rely on the immutability of the data and content-addressability via hashing for storing and accessing the data, but these data cannot be modified after their creation. If they could be modified, the content-based key assigned to each object at its creation would be rather insignificant semantically, because it would no longer correspond to the hash digest of the content of that object after the modification. The most important consequence of allowing this behavior, though, is that such modifications on an object should be applied on all Agents that store replicas of that object in a certain order – the same order across all of them – in order to guarantee the consistency of the data stored in the systems.

  We should pick a consistency model for the systems, perhaps a strong one, such as *linearizability* or *sequential consistency*, so as to be in the same spirit as the already supported *read-after-create* consistency, although this is not really necessary. *Eventual consistency* could also be used for Update operations along with a strong consistency model for Create operations – this is pretty much the behavior of Amazon S3[33].

One possible extension of `MICAS` and `RICAS` in this direction would be to develop an additional layer on top of them to implement object versioning. In this case, each object would be given a different name, possibly by the client, and multiple subsequent versions of it would be managed by the additional layer, probably involving an algorithm to implement consensus, e.g. Raft, whereas the data of the versions themselves would still be stored in the underlying `MICAS` or `RICAS`, thus still leveraging data immutability and content-addressability via hashing.

- **Delete operations**

  Of course, no system can be considered to be complete unless it supports Delete operations on the stored data. However, as we have mentioned in chapter 3, the Delete operation is effectively a way to mutate the state stored; hence, it is governed by the same principles as the Update operation that was discussed above. In fact, many storage systems implement Delete operation on an object as an Update to a special "tombstone version" of it.

Bibliography & References

[1] Docker project, https://www.docker.com/what-docker, accessed on the April 24th, 2018.

[2] fleet - a distributed init system, https://github.com/coreos/fleet/, accessed on the April 24th, 2018.

[3] CoreOS Container Linux, https://coreos.com/os/docs/latest/, accessed on the April 24th, 2018.

[4] Kubernetes, https://kubernetes.io/, accessed on April 24th, 2018.

[5] Tectonic, https://coreos.com/tectonic/, accessed on the April 24th, 2018.

[6] Swarm: a Docker-native clustering system, https://github.com/docker/swarm, accessed on the April 24th, 2018.

[7] OpenShift, https://www.openshift.com/, accessed on the April 24th, 2018.

[8] Apache Mesos, https://mesos.apache.org/, accessed on the April 24th, 2018.

[9] DC/OS, https://dcos.io/, accessed on the April 24th, 2018.

[10] Thomas Erl, *SOA: Principles of Service Design*, Prentice Hall, 2007.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter

Vosshall and Werner Vogels, *Dynamo: Amazon's Highly Available Key-value Store*, Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles (SOSP '07), Washington, US, October 2007.

[12] Docker documentation, *About storage drivers - Containers and layers*, `https://docs.docker.com/storage/storagedriver/#container-and-layers`, accessed on the April 24th, 2018.

[13] Alex Chircop, *Persistent Storage for Containers: Stateful Apps in Docker*, February 2018, `https://storageos.com/persistent-storage-containers-stateful-apps-docker/`, accessed on the April 24th, 2018.

[14] Portworx, *2017 Annual Container Adoption Survey: Huge Growth in Containers*, April 2017, `https://portworx.com/2017-container-adoption-survey/`, accessed on the April 24th, 2018.

[15] Adam Wiggins, The Twelve-Factor App, `https://12factor.net/`, accessed on the April 24th, 2018.

[16] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, *Ceph: A Scalable, High-Performance Distributed File System*, Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06), November 2006.

[17] S. A. Weil, A. W. Leung, S. A. Brandt, C. Maltzahn, *RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters*, Petascale Data Storage Workshop SC07, November 2007.

[18] S. Weil, S. A. Brandt, E. L. Miller, C. Maltzahn, *CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data*, Proceedings of SC '06, November 2006.

[19] A. Lakshman, P. Malik, *Cassandra - A Decentralized Structured Storage System*, appeared at 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS 2009), Big Sky, MT, US, October 2009.

[20] Jonathan Ellis, *Facebook's Cassandra paper, annotated and compared to Apache Cassandra 2.0*, `https://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html`, accessed on the April 26th, 2018.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, Daniel Lewin, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*, Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, p. 654-663, May 1997.

[22] Swift Architectural Overview, `https://docs.openstack.org/swift/latest/overview_architecture.html`, accessed on the April 24th, 2018.

[23] Why Riak KV?, `http://docs.basho.com/riak/kv/2.2.3/learn/why-riak-kv/`, accessed on the April 24th, 2018.

[24] Project Voldemort documentation, Design, `http://www.project-voldemort.com/voldemort/design.html`, accessed on the April 24th, 2018.

[25] GlusterFS Algorithms: Distribution, `https://www.gluster.org/glusterfs-algorithms-distribution/`, March 2012, accessed on the April 24th, 2018.

[26] Torus Distributed Storage, `https://github.com/coreos/torus`, accessed on the May 1st, 2018.

[27] Barak Michener, *Presenting Torus: A modern distributed storage system by CoreOS*, June 2016, `https://coreos.com/blog/torus-distributed-storage-by-coreos.html`, accessed on the May 1st, 2018.

[28] etcd, `https://github.com/coreos/etcd/`, accessed on the May 3rd, 2018.

[29] Barak Michener, *Torus: Focusing Storage for Kubernetes by Barak Michener, CoreOS, Inc.*, Talk at KubeCon 2016, `https://youtu.be/f8Ipew7JYFU`, accessed on the May 3rd, 2018.

[30] gRPC, A high performance, open-source universal RPC framework, `https://grpc.io/`, accessed on the May 3rd, 2018.

[31] I. S. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields*, Journal of the Society for Industrial and Applied Mathematics (SIAM), 1960.

[32] Rook - documentation, https://rook.github.io/docs/rook/master/, accessed on the April 24th, 2018.

[33] Amazon S3, https://aws.amazon.com/s3/, accessed on the May 16th, 2018.

[34] Randy Bias, *The History of Pets vs Cattle and How to Use the Analogy Properly*, September 2016, http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/, accessed on the April 26th, 2018.

[35] Kubernetes documentation, *Example: Deploying Cassandra with Stateful Sets*, https://kubernetes.io/docs/tutorials/stateful-application/cassandra/, accessed on the April 27th, 2018.

[36] Minikube, https://github.com/kubernetes/minikube, accessed on the April 27th, 2018.

[37] Animesh Singh, Anthony Amanse, Ishan Gulhane, *Deploy a scalable Apache Cassandra database on Kubernetes*, March 2017, https://developer.ibm.com/code/patterns/deploy-a-scalable-apache-cassandra-database-on-kubernetes/, accessed on the April 28th, 2018.

[38] Portworx, *White Paper: A New Storage Architecture For The Commoditization Era*, http://portworx.com/wp-content/uploads/2017/06/Portworx_Tech_Report_6-14-16.pdf, accessed on the April 27th, 2018.

[39] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D., *Epidemic algorithms for replicated database maintenance*, Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), Vancouver, British Columbia, Canada, pp. 1–12. ACM, New York (1987).

[40] Portworx documentation, A Production Ops Guide to Deploying Cassandra, https://docs.portworx.com/applications/cassandra.html, accessed on the April 28th, 2018.

[41] StorageOS, *StorageOS – Platform Architecture Overview*, http://resources.storageos.com/storageos_platform_architecture_overview, retrieved on April 28th, 2018.

[42] StorageOS documentation, Cassandra with StorageOS, https://docs.storageos.com/docs/applications/cassandra, accessed on the April 28th, 2018.

[43] Arrikto, Inc., *Rok: Decentralized storage for the cloud native world*, February 2018, http://arrikto.com/wp-content/uploads/2018/03/20180206-rok_decentralized_storage_for_the_cloud_native_world.pdf, accessed on the April 28th, 2018.

[44] Apache Mail Archives, Subject: *Cassandra storage: Some thoughts*, March 2018, https://lists.apache.org/thread.html/d5eb8608cb1ccfcf3a173238b5b981cb539ff1a356ef158a345b9308@%3Cuser.cassandra.apache.org%3E, accessed on the April 28th, 2018.

[45] Chris Pavlou, *Why your Cassandra needs local NVMe and Rok*, January 2018, https://journal.arrikto.com/why-your-cassandra-needs-local-nvme-and-rok-1787b9fc286d, accessed on the April 28th, 2018.

[46] Wikipedia – The Free Encyclopedia, *Operating-system-level virtualization*, https://en.wikipedia.org/wiki/Operating-system-level_virtualization, accessed on the 11th August, 2018.

[47] The Single UNIX Specification, Version 2, 1997, The Open Group, *chroot - change root directory*, http://pubs.opengroup.org/onlinepubs/7908799/xsh/chroot.html, accessed on the 11th August, 2018.

[48] Linux manual pages, namespaces(7), http://man7.org/linux/man-pages/man7/namespaces.7.html, accessed on the 12th August, 2018.

[49] Linux manual pages, proc(5), man7.org/linux/man-pages/man5/proc.5.html, accessed on the 12th August, 2018.

[50] Linux manual pages, clone(2), http://man7.org/linux/man-pages/man2/clone.2.html, accessed on the 12th August, 2018.

[51] Linux manual pages, unshare(2), man7.org/linux/man-pages/man2/unshare.2.html, accessed on the 12th August, 2018.

[52] Linux manual pages, setns(2), http://man7.org/linux/man-pages/man2/setns.2.html, accessed on the 12th August, 2018.

[53] Linux manual pages, mount(2), http://man7.org/linux/man-pages/man2/mount.2.html, accessed on the 12th August, 2018.

[54] Linux manual pages, mount(8), http://man7.org/linux/man-pages/man8/mount.8.html, accessed on the 12th August, 2018.

[55] Michael Kerrisk, *Namespaces in operation, part 1: namespaces overview*, Linux Weekly News, January 4, 2013, https://lwn.net/Articles/531114/, accessed on the 12th August, 2018.

[56] Linux manual pages, chroot(2), man7.org/linux/man-pages/man2/chroot.2.html, accessed on the 12th August, 2018.

[57] Linux manual pages, path_resolution(7), http://man7.org/linux/man-pages/man7/path_resolution.7.html, accessed on the 12th August, 2018.

[58] The Linux Kernel Archives – Documentation, *Shared Subtrees*, https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt, accessed on the 12th August, 2018.

[59] Linux manual pages, systemd(1), http://man7.org/linux/man-pages/man1/init.1.html, accessed on the 12th August, 2018.

[60] Linux manual pages, user_namespaces(7), http://man7.org/linux/man-pages/man7/user_namespaces.7.html, accessed on the 12th August, 2018.

[61] Linux manual pages, cgroups(7), http://man7.org/linux/man-pages/man7/cgroups.7.html, accessed on the 13th August, 2018.

[62] Linux kernel documentation, *Complete Fairness Queueing* https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt, accessed on the 14th August, 2018.

[63] Debian Wiki, *Hugepages*, https://wiki.debian.org/Hugepages, accessed on the 14th August, 2018.

[64] Wikipedia – The Free Encyclopedia, *Remote direct memory access*, https://en.wikipedia.org/wiki/Remote_direct_memory_access, accessed on the 14th August, 2018.

[65] Wikipedia – The Free Encyclopedia, *InfiniBand*, https://en.wikipedia.org/wiki/InfiniBand, accessed on the 14th August, 2018.

[66] Goldwyn Rodrigues, *Unifying filesystems with union mounts*, Linux Weekly News, December 24, 2008, https://lwn.net/Articles/312641/, accessed on the 14th August, 2018.

[67] Valerie Aurora (formerly Henson), *Unioning file systems: Architecture, features, and design choices*, Linux Weekly News, March 18, 2009, https://lwn.net/Articles/324291/, accessed on the 14th August, 2018.

[68] *Unionfs: A Stackable Unification File System*, http://unionfs.filesystems.org/, accessed on the 14th August, 2018.

[69] aufs – advanced multi-layered unification filesystem, http://aufs.sourceforge.net/, accessed on the 14th August, 2018.

[70] Neil Brown, *Overlay Filesystem*, Linux kernel documentation, https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt, accessed on the 14th August, 2018.

[71] Nigel Brown, *The Overlay Filesystem*, https://windsock.io/the-overlay-filesystem/, accessed on the 15th August, 2018.

[72] Docker documentation, *Docker – Get Started*, https://docs.docker.com/get-started, accessed on the 16th August, 2018.

[73] Docker documentation, *Docker Overview*, https://docs.docker.com/engine/docker-overview/, accessed on the 16th August, 2018.

[74] Jonathan Corbet, *Process containers*, Linux Weekly News, May 29, 2007, https://lwn.net/Articles/236038/, accessed on the 18th August, 2018.

[75] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, John Wilkes, *Borg, Omega, and Kubernetes – Lessons learned from three container-management systems over a decade*, March 2016, ACM Queue, Volume 14, pp. 70-93, https://ai.google/research/pubs/pub44843, accessed on the 18th August, 2018.

[76] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes, *Large-scale cluster management at Google with Borg*, 2015, Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France, https://ai.google/research/pubs/pub43438

[77] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, John Wilkes, *Omega: flexible, scalable schedulers for large compute clusters*, 2013, SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic, pp. 351-364, https://ai.google/research/pubs/pub41684, accessed on the 18th August, 2018.

[78] Kubernetes documentation – Concepts, *Pod Overview*, https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, accessed on the August 19th, 2018.

[79] Kubernetes documentation – Concepts, *Labels and Selectors* https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/, accessed on the August 20th, 2018.

[80] Kubernetes documentation – Concepts, *ReplicationController*, https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/, accessed on the August 20th, 2018.

[81] Kubernetes documentation – Concepts, *ReplicaSet*, https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, accessed on the August 20th, 2018.

[82] Kubernetes documentation – Concepts, *Deployments*, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, accessed on the August 20th, 2018.

[83] Kubernetes documentation – Concepts, *StatefulSets*, https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, accessed on the August 20th, 2018.

[84] Kubernetes documentation – Concepts, *Services*, https://kubernetes.io/docs/concepts/services-networking/service/, accessed on the August 20th, 2018.

[85] Kubernetes documentation – Concepts, *Concepts Underlying the Cloud Controller Manager*, https://kubernetes.io/docs/concepts/architecture/cloud-controller/, accessed on the August 21st, 2018.

[86] Kubernetes documentation – Concepts, *Kubernetes Components*, , accessed on the August 21st, 2018.

[87] *Open Container Initiative*, https://www.opencontainers.org/, accessed on the August 21st, 2018.

[88] Andrew Tridgell, *Efficient Algorithms for Sorting and Synchronization*, PhD dissertation, February 1999, Australian National University.

[89] Wikipedia – The Free Encyclopedia, *Rsync*, https://en.wikipedia.org/wiki/Rsync, accessed on the 28th July, 2018.

[90] *How Rsync Works – A Practical Overview*, https://rsync.samba.org/how-rsync-works.html, accessed on the 28th July, 2018.

[91] Linux manual pages, rsync(1), http://man7.org/linux/man-pages/man1/rsync.1.html, accessed on the 28th July, 2018.

[92] Linux manual pages, rsyncd.conf(5), http://man7.org/linux/man-pages/man5/rsyncd.conf.5.html, accessed on the 28th July, 2018.

[93] Wikipedia – The Free Encyclopedia, *Adler-32*, https://en.wikipedia.org/wiki/Adler-32, accessed on the 28th July, 2018.

[94] Wikipedia – The Free Encyclopedia, *Create, read, update and delete*, https://en.wikipedia.org/wiki/Create,_read,_update_and_delete, accessed on the May 28th, 2018.

[95] George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair, *Distributed Systems – Concepts and Design*, 5th Edition, 2011, Pearson Higher Education.

[96] Andrew S. Tanenbaum, Maarten Van Steen, *Distributed Systems – Principles and Paradigms*, 2nd Edition, 2006, Prentice Hall.

[97] James F. Kurose, Keith W. Ross, *Computer Networking: A Top-Down Approach*, 6th Edition, 2013, Pearson Education, Addison-Wesley.

[98] T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, RFC 1945, published by the Internet Society on behalf of the Internet Engineering Task Force (IETF), May 1996, `https://tools.ietf.org/html/rfc1945`, accessed on the June 1st, 2018.

[99] R. Fielding, J. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, RFC 7230, published by the Internet Society on behalf of the Internet Engineering Task Force (IETF), June 2014, `https://tools.ietf.org/html/rfc7230`, accessed on the June 1st, 2018.

[100] M. Belshe, R. Peon, M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540, published by the Internet Society on behalf of the Internet Engineering Task Force (IETF), May 2015, `https://tools.ietf.org/html/rfc7540`, accessed on the June 1st, 2018.

[101] The Chromium Projects – SPDY, *SPDY: An experimental protocol for a faster web*, `https://www.chromium.org/spdy/spdy-whitepaper`, accessed on the June 1st, 2018.

[102] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, PhD dissertation, 2000, University of California, Irvine.

[103] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, published by the Internet Society on behalf of the Internet Engineering Task Force (IETF), December 2017, `https://tools.ietf.org/html/rfc8259`, accessed on the June 1st, 2018.

[104] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 4648, published by the Internet Society on behalf of the Internet Engineering Task Force

(IETF), October 2006, `https://tools.ietf.org/html/rfc4648`, accessed on the June 1st, 2018.

[105] Andrew D. Birrell, Bruce Jay Nelson, *Implementing Remote Procedure Calls*, Journal ACM Transactions on Computer Systems (TOCS) Volume 2 Issue 1, February 1984, pages 39-59.

[106] Protocol Buffers – Developer Guide, `https://developers.google.com/protocol-buffers/docs/overview`, accessed on the June 3rd, 2018.

[107] Pat Helland, *Data on the Outside versus Data on the Inside*, 2005, Proceedings of the 2005 CIDR Conference (Conference on Innovative Database Research).

[108] Pat Helland, *Immutability Changes Everything*, 7th Biennial Conference on Innovative Data Systems Research (CIDR'15) January 4-7, 2015, Asilomar, California, USA.

[109] Wikipedia – The Free Encyclopedia, *Persistent data structure*, `https://en.wikipedia.org/wiki/Persistent_data_structure`, accessed on the May 4th, 2018.

[110] Git, `https://git-scm.com/`, accessed on the May 5th, 2018.

[111] Philip Nilsson, *Git is a purely functional data structure*, March 2013, `https://blog.jayway.com/2013/03/03/git-is-a-purely-functional-data-structure/`, accessed on the May 5th, 2018.

[112] Martin Kleppmann, *Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, Published by O'Reilly Media, Inc.

[113] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, *Database System Concepts*, 6th Edition, 2010, McGraw-Hill Education.

[114] by Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, 7th Edition, 2015, Pearson.

[115] Berkeley DB Programmer's Reference Guide, Chapter 11: Berkeley DB Transactional Data Store Applications – Berkeley DB recoverability, https://docs.oracle.com/cd/E17276_01/html/programmer_reference/transapp_reclimit.html, accessed on the May 7th, 2018.

[116] Jay Kreps, *The Log: What every software engineer should know about real-time data's unifying abstraction*, December 2013, https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying, accessed on the May 7th, 2018.

[117] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: a tutorial.*, ACM Computing Surveys 22, 4 (Dec. 1990), 299–319.

[118] Leslie Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21, 7 (July 1978), 558-565.

[119] Leslie Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.

[120] Leslie Lamport, *Paxos Made Simple*, ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.

[121] Leslie Lamport, *Fast Paxos*, Distributed Computing 19, 2 (October 2006) 79-103.

[122] Leslie Lamport, *Generalized Consensus and Paxos*, Microsoft Research Technical Report MSR-TR-2005-33 (15 March 2005).

[123] Leslie Lamport and Mike Massa, *Cheap Paxos*, Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004) held in Florence in June-July 2004.

[124] Miguel Castro, Barbara Liskov, *Practical Byzantine Fault Tolerance*, (February 1999), Proceedings of the Third Symposium on Operating Systems Design and Implementation: 173–186.

[125] Diego Ongaro and John Ousterhout, *In Search of an Understandable Consensus Algorithm*, Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference, Philadelphia, USA, June 2014.

[126] Flavio P. Junqueira, Benjamin C. Reed, Marco Serafin, *Zab: High-performance broadcast for primary-backup systems*, DSN '11 Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks, pages 245-256.

[127] Barbara Liskov, James Cowling, *Viewstamped Replication Revisited*, Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC '88), pages 8-17, Toronto, Ontario, Canada, August 1988.

[128] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987.

[129] Wikipedia – The Free Encyclopedia, *Snapshot isolation*, `https://en.wikipedia.org/wiki/Snapshot_isolation`, accessed on the May 9th, 2018.

[130] Patrick O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth O'Neil, *The log-structured merge-tree (LSM-tree)*, Acta Informatica, Volume 33, Issue 4, June 1996.

[131] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, 7th USENIX Symposium on Operating System Design and Implementation (OSDI), November 2006.

[132] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd edition, MIT Press, 2009.

[133] William Pugh, *Skip Lists: A Probabilistic Alternative to Balanced Trees*, Magazine Communications of the ACM Volume 33 Issue 6, June 1990, pages 668-676.

[134] Apache HBase, `https://hbase.apache.org/`, accessed on the May 13th, 2018.

[135] LevelDB, `https://github.com/google/leveldb`, accessed on the May 11th, 2018.

[136] RocksDB, https://rocksdb.org/, accessed on the May 11th, 2018.

[137] WiredTiger architecture, http://source.wiredtiger.com/3.0.0/architecture.html, accessed on the May 12th, 2018.

[138] Alan Morrison, *The rise of immutable data stores*, http://usblogs.pwc.com/emerging-technology/the-rise-of-immutable-data-stores/, accessed on the May 14th, 2018.

[139] Apache Kafka, https://kafka.apache.org/, accessed on the May 14th, 2018.

[140] Apache Kafka – documentation, https://kafka.apache.org/documentation/, accessed on the May 14th, 2018.

[141] Apache Samza, https://samza.apache.org/, accessed on the May 14th, 2018.

[142] NATS Server, https://nats.io/, accessed on the May 14th, 2018.

[143] Amazon Kinesis, https://aws.amazon.com/kinesis/, accessed on the May 14th, 2018.

[144] Martin Kleppmann, *Turning the database inside-out with Apache Samza*, March 2015, https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html, accessed on the May 14th, 2018.

[145] Datomic Cloud Documentation, https://docs.datomic.com/cloud/index.html, accessed on the May 16th, 2018.

[146] Rich Hickey, *Deconstructing the Database*, talk at JaxConf 2012, https://youtu.be/Cym4TZwTCNU, accessed on the May 16th, 2018.

[147] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, Aviad Zuck, *Tango: Distributed Data Structures over a Shared Log*, SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 325-340, Pennsylvania, US, November 2013.

[148] CenturyLink – Developer Center – Blog (originally posted on the Orchestrate blog), *Immutability and Why It Matters*, https://www.ctl.io/developers/blog/post/immutability, accessed on the May 18th, 2018.

[149] Apache CouchDB, https://couchdb.apache.org/, accessed on the May 21st, 2018.

[150] RethinkDB, https://www.rethinkdb.com/, accessed on the May 20th, 2018.

[151] Leif Walsh, Vyacheslav Akhmechet, Mike Glukhovs, Hexagram 49, Inc., *RethinkDB — Rethinking Database Storage*, July 2009.

[152] Baron Schwartz, *Immutability, MVCC, and Garbage Collection*, December 2013, https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/, accessed on the May 20th, 2018.

[153] Wikipedia – The Free Encyclopedia, *Content-addressable storage*, https://en.wikipedia.org/wiki/Content-addressable_storage, accessed on the May 21st, 2018.

[154] Wikipedia – The Free Encyclopedia, *Hash function*, https://en.wikipedia.org/wiki/Hash_function, accessed on the May 22nd, 2018.

[155] Wikipedia – The Free Encyclopedia, *Cryptographic hash function*, https://en.wikipedia.org/wiki/Cryptographic_hash_function, accessed on the May 22nd, 2018.

[156] NIST, *Descriptions of SHA-256, SHA-384, and SHA-512*, https://web.archive.org/web/20130526224224/http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf, accessed on the May 25th, 2018.

[157] Sean Quinlan, Sean Dorward, *Venti: a new approach to archival storage*, Proceedings of the FAST 2002 Conference on File and Storage Technologies, pp. 89–102, January 2002.

[158] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas Bressoud, Adrian Perrig, *Opportunistic Use of Content Addressable Storage for*

Distributed File Systems, Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 2003.

[159] Wikipedia – The Free Encyclopedia, *Discrete uniform distribution*, https://en.wikipedia.org/wiki/Discrete_uniform_distribution, accessed on the May 30th, 2018.

[160] Athanasios Papoulis and S. Unnikrishna Pillai, *Probability, Random Variables and Stochastic Processes*, 4th Edition, McGraw-Hill Europe, 2002.

[161] Wikipedia – The Free Encyclopedia, *Birthday problem*, https://en.wikipedia.org/wiki/Birthday_problem, accessed on the May 23rd, 2018.

[162] Jim Gray, Catharine van Ingen, *Empirical Measurements of Disk Failure Rates and Error Rates*, Microsoft Research Technical Report MSR-TR-2005-166, December 2005.

[163] Wikipedia – The Free Encyclopedia, *Birthday problem*, https://en.wikipedia.org/wiki/Birthday_attack, accessed on the May 24th, 2018.

[164] National Institute of Standards and Technology, *Federal Information Processing Standards Publication (FIPS PUB) 180-4. Secure Hash Standard (SHS)*, US Department of Commerce, August 2015.

[165] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, Yarik Markov, *The first collision for full SHA-1*, https://shattered.io/, accessed on the May 26th, 2018.

[166] Wikipedia – The Free Encyclopedia, *MD5*, https://en.wikipedia.org/wiki/MD5, accessed on the May 26th, 2018.

[167] CERT Division, Software Engineering Institute, Carnegie Mellon University, *MD5 vulnerable to collision attacks – CERT Vulnerability Note VU#836068*, December 2008, https://www.kb.cert.org/vuls/id/836068, accessed on the May 26th, 2018.

[168] Wikipedia – The Free Encyclopedia, *MurmurHash*, https://en.wikipedia.org/wiki/MurmurHash, accessed on the May 27th, 2018.

[169] Google's CityHash code repository, https://github.com/google/cityhash, accessed on the May 27th, 2018.

[170] Google's FarmHash code repository, https://github.com/google/farmhash, accessed on the May 27th, 2018.

[171] J. Alakuijala, B. Cox, J. Wassenber, *Fast keyed hash/pseudo-random function using SIMD multiply and permute*, arXiv, February 2017, https://arxiv.org/pdf/1612.06257.pdf, accessed on the May 27th, 2018.

[172] Google's HighwayHash code repository, https://github.com/google/highwayhash/, accessed on the May 27th, 2018.

[173] Jean-Philippe Aumasson, Daniel J. Bernstein, Martin Boßlet, *Hash-flooding DoS reloaded: attacks and defenses* December 2012, https://131002.net/siphash/siphashdos_29c3_slides.pdf, accessed on the May 27th, 2018.

[174] HighwayHash implementation by Minio, Inc., https://github.com/minio/highwayhash, accessed on the May 27th, 2018.

[175] Frank Wessels, *HighwayHash: Fast hashing at over 10 GB/s per core in Golang*, January 2018, https://blog.minio.io/highwayhash-fast-hashing-at-over-10-gb-s-per-core-in-golang-fee938b5218a, accessed on the May 27th, 2018.

[176] *BLAKE2 — fast secure hashing*, https://blake2.net/, accessed on the May 27th, 2018.

[177] Wikipedia – The Free Encyclopedia, *Partition (database)*, https://en.wikipedia.org/wiki/Partition_(database), accessed on the May 29th, 2018.

[178] Wikipedia – The Free Encyclopedia, *Shard (database architecture)*, https://en.wikipedia.org/wiki/Shard_(database_architecture), accessed on the May 29th, 2018.

[179] Wikipedia – The Free Encyclopedia, *Shared-nothing architecture*, https://en.wikipedia.org/wiki/Shared-nothing_architecture, accessed on the May 29th, 2018.

[180] Kubernetes documentation – Concepts, *Assigning Pods to Nodes*, https://kubernetes.io/docs/concepts/configuration/assign-pod-node/, accessed on the June 6th, 2018.

[181] Wikipedia – The Free Encyclopedia, *Balls into bins*, https://en.wikipedia.org/wiki/Balls_into_bins, accessed on the June 13th, 2018.

[182] Norman L. Johnson and Samuel Kotz, *Urn Models and Their Application – An Approach to Modern Discrete Probability Theory*, published by John Wiley & Sons, 1977.

[183] Kubernetes Community on GitHub, *API Conventions*, https://github.com/kubernetes/community/blob/master/contributors/devel/api-conventions.md, accessed on the June 18th, 2018.

[184] Kubernetes documentation, *Building Large Clusters*, https://kubernetes.io/docs/admin/cluster-large/, accessed on the June 19th, 2018.

[185] flannel, https://github.com/coreos/flannel, accessed on the 21st June, 2018.

[186] Kubernetes documentation – Concepts, *Volumes*, https://kubernetes.io/docs/concepts/storage/volumes, accessed on the 18th July, 2018.

[187] Kubernetes documentation – Concepts, *Persistent Volumes*, https://kubernetes.io/docs/concepts/storage/persistent-volumes, accessed on the 18th July, 2018.

[188] MySQL 8.0 Reference Manual – Alternative Storage Engines, *The MyISAM Storage Engine* https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html, accessed on the 24th July, 2018.

[189] MySQL 8.0 Reference Manual, *The InnoDB Storage Engine*, https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html, accessed on the 24th July, 2018.

[190] Daniel Bovet, Marco Cesati, *Understanding the Linux Kernel*, 3rd Edition, 2008, O'Reilly Media.

[191] Michael Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 2010, No Starch Press.

[192] Richard Crowley, *Things UNIX can do atomically*, https://rcrowley.org/2010/01/06/things-unix-can-do-atomically.html, accessed on the 25th July, 2018.

[193] Linux manual pages, fsync(2), fdatasync(2), http://man7.org/linux/man-pages/man2/fdatasync.2.html, accessed on the 25th July, 2018.

[194] Paul McKenney, *What is RCU, Fundamentally?*, Linux Weekly News, December 17, 2007, https://lwn.net/Articles/262464/, accessed on the 27th July, 2018.

[195] Maurice Herlihy, Nir Shavit, *The Art of Multiprocessor Programming*, 2008, Elsevier.

[196] Pedro Ramalhete, *Lock-Free and Wait-Free, definition and examples*, May 2013, https://concurrencyfreaks.blogspot.com/2013/05/lock-free-and-wait-free-definition-and.html, accessed on the 27th July, 2018.

[197] Linux manual pages, *vfork(2)*, http://man7.org/linux/man-pages/man2/vfork.2.html, accessed on the 31st July, 2018.

[198] SkyDNS – DNS service discovery for etcd, https://github.com/skynetservices/skydns, accessed on the August 1st, 2018.

[199] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[200] etcd Operator, https://github.com/coreos/etcd-operator, accessed on the August 8th, 2018.

[201] kafka-operator – A Kafka Operator for Kubernetes, https://github.com/krallistic/kafka-operator, accessed on the August 8th, 2018.

[202] Strimzi: Kafka as a Service, https://github.com/strimzi/strimzi-kafka-operator, accessed on the August 8th, 2018.

[203] PostgreSQL Operator, https://github.com/CrunchyData/postgres-operator, accessed on the August 8th, 2018.

[204] Postgres Operator, https://github.com/zalando-incubator/postgres-operator, accessed on the August 8th, 2018.

[205] Prototype Kubernetes Operator for CouchDB, https://github.com/nicolai86/couchdb-operator, accessed on the August 8th, 2018.

[206] MySQL Operator, https://github.com/oracle/mysql-operator, accessed on the August 8th, 2018.

[207] Cassandra Operator – Kubernetes Operator for Apache Cassandra, https://github.com/instaclustr/cassandra-operator, accessed on the 8th August, 2018.

[208] https://github.com/operator-framework, accessed on the August 8th, 2018.