ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Ενορχήστρωση Συστημάτων Αποθήκευσης Δεδομένων με χρήση Χειριστών στο περιβάλλον Kubernetes: Μελέτη Περίπτωσης βασισμένη στην Cassandra

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Χ. Ζαρκάδας
Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης

Αθήνα, Φεβρουάριος 2019

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

Orchestrating Stateful Workloads using Kubernetes Operators: A Cassandra Case Study

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Χ. Ζαρκάδας
Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Μαρτίου 2019.

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

Αθήνα, Φεβρουάριος 2019

. . . . . .

Ιωάννης Χ. Ζαρκάδας
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Η διαχείριση εφαρμογών αποθήκευσης δεδομένων ήταν ανέκαθεν ένα δύσκολο και ακριβό πρόβλημα. Οι εφαρμογές αποθήκευσης δεδομένων και συγκεκριμένα οι κατανεμημένες βάσεις δεδομένων, απαιτούν ιδιαίτερη προσοχή στη διαχείρισή τους, καθώς περιέχουν πολίτιμα δεδομένα και λάθος χειρισμοί μπορούν να αποτελέσουν αιτία απώλειάς τους. Δεν είναι τυχαίο ότι έχουν αναπτυχθεί πολλά επιχειρηματικά μοντέλα γύρω από την παροχή υπηρεσιών αποθήκευσης δεδομένων χωρίς την ανάγκη διαχείρισης του συστήματος, το λεγόμενο Software as a Service (SaaS).

Στο πλαίσιο αυτό, η Apache Cassandra είναι μία κατανεμημένη βάση δεδομένων, η οποία χρησιμοποιείται με επιτυχία, εδώ και πολλά χρόνια, σε production περιβάλλοντα (πχ Netflix). Η διαχείρισή της όμως, παραμένει δύσκολη και απαιτεί προσεκτικές κινήσεις και καλή γνώση του διαχειριστή. Έχουν υπάρξει προσπάθειες αυτοματοποίησης της διαχείρισης της Cassandra, αλλά μέχρι τώρα είναι είτε κλειστό εμπορικό λογισμικό, είτε δουλεύουν μόνο σε συγκεκριμένους Cloud Providers.

Από την άλλη, ο Kubernetes είναι μία πλατφόρμα διαχείρισης εργασιών σε Containers, που τρέχει σε οποιοδήποτε περιβάλλον (Cloud, On-Prem) και προσφέρει αρκετή έτοιμη λειτουργικότητα για διαχείριση εφαρμογών αποθήκευσης δεδομένων.

Θέλουμε λοιπόν να φτιάξουμε ένα λογισμικό διαχείρισης της Apache Cassandra, το οποίο θα είναι ανοιχτού κώδικα και θα μπορεί να λειτουργήσει σε οποιοδήποτε περιβάλλον. Για τον λόγο αυτό, θα προσπαθήσουμε να αναπτύξουμε το λογισμικό διαχείρισης της Cassandra πάνω στην πλατφόρμα Kubernetes. Θα αξιολογήσουμε τις έτοιμες λύσεις που προσφέρει ο Kubernetes για τη διαχείριση λογισμικού αποθήκευσης δεδομένων, θα δούμε γιατί δεν μπορούν να χρησιμοποιηθούν όπως είναι για την Cassandra και θα τις επεκτείνουμε ώστε να τρέξουμε μια λειτουργική Cassandra με πλήρως αυτοματοποιημένη διαχείριση.

Λέξεις-Κλειδιά

kubernetes, cassandra, operator, κατανεμημένα συστήματα, συστήματα αποθήκευσης δεδομένων, κατανεμημένη αποθήκευση, containers

Abstract

In this thesis, we present the design and implementation of *Cassandra-Operator*, a management layer for Apache Cassandra which leverages the Kubernetes platform to achieve cross-cloud compatibility and benefit from existing Kubernetes primitives. The proposed design is based on the *Operator Pattern* [1], in which the management layer uses the Kubernetes API to control the infrastucture and the etcd database of Kubernetes for persistent storage. Our case focuses on Apache Cassandra, because of its popularity, difficulty to manage and lack of good open-source solutions.

The *Cassandra Operator* leverages the best-of-breed patterns to provide Cassandra with automated deployment, scaling, self-healing, monitoring, backups and restores. In the last years, the amount of data produced has grown exponentially into unprecedented scale. This brought the need for scalable storage systems, capable of handling the ever-growing amount of new data and enable users to quickly query, process and analyze them. A new paradigm emerged, one that was designed to run in a distributed and fault-tolerant way. That paradigm was NoSQL and it has since made huge advances, both in scientific literature as well as real-world systems. While the technology powering these systems is mature, there is still much manual work involved in managing such a complex system.

Each database requires its own set of actions to perform the creation of a new cluster, scaling up or down, replacing a dead member, monitoring, health checks, backups and restores. In addition, those actions require provisioning and management of machines and because each cloud provider has its own commands for these actions, it means management software made for one cloud-provider will not work on another. Kubernetes is a platform that abstracts the underlying infrastructure and also provides us with helpful primitives to run stateful workloads. Kubernetes gives us the ability to develop a management layer, called a Kubernetes Operator, which will run as a regular Kubernetes workload, interface with the Kubernetes API to leverage built-in primitives and use the Kubernetes etcd database for persistent storage.

Keywords

Kubernetes, Cassandra, Operators, Controllers, stateful systems, distributed storage, orchestration, containers

Αντί Προλόγου

Στο σημείο αυτό θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς τους ανθρώπους που συνέδραμαν στην ολοκλήρωση αυτής της διπλωματικής εργασίας, αλλά και στην ευρύτερη ακαδημαϊκή μου πορεία. Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη, που μέσα από τις διαλέξεις του καλλιέργησε το ενδιαφέρον μου για τον τομέα των Υπολογιστικών Συστημάτων. Επίσης, ευχαριστώ θερμά τον διδάκτορα Βαγγέλη Κούκη για την εμπιστοσύνη που μου έδειξε, για τον μεταδοτικό ενθουσιασμό του και τις πολύτιμες γνώσεις που αποκόμισα κατά τη διάρκεια της εκπόνησης αυτής της εργασίας χάρη στη βοήθειά του. Επιπλέον, θέλω να εκφράσω την ευγνωμοσύνη μου προς όσες και όσους υποστηρίζουν έμπρακτα την ελεύθερη και δωρεάν διακίνηση της γνώσης, αναφέροντας χαρακτηριστικά τη συνδρομή της Arrikto, της κοινότητας του Rook και της ScyllaDB. Τέλος, ευχαριστώ από καρδιάς την οικογένειά μου και τους αγαπημένους μου φίλους για τη στήριξη, την κατανόηση και την ανεκτίμητη συντροφιά τους.

Ιωάννης Ζαρκάδας
Μάρτιος 2019

Contents

List of figures

List of tables

0.1 Εισαγωγή

0.1.1 Σκοπός & Κίνητρο

Η διαχείριση εφαρμογών αποθήκευσης δεδομένων ήταν ανέκαθεν ένα δύσκολο και ακριβό πρόβλημα. Οι εφαρμογές αποθήκευσης δεδομένων και συγκεκριμένα οι κατανεμημένες βάσεις δεδομένων, απαιτούν ιδιαίτερη προσοχή στη διαχείρισή τους, καθώς περιέχουν πολίτιμα δεδομένα και λάθος χειρισμοί μπορούν να αποτελέσουν αιτία απώλειάς τους. Δεν είναι τυχαίο ότι έχουν αναπτυχθεί πολλά επιχειρηματικά μοντέλα γύρω από την παροχή υπηρεσιών αποθήκευσης δεδομένων χωρίς την ανάγκη διαχείρισης του συστήματος, το λεγόμενο Software as a Service (SaaS).

Στο πλαίσιο αυτό, η Apache Cassandra είναι μία κατανεμημένη βάση δεδομένων, η οποία χρησιμοποιείται με επιτυχία, εδώ και πολλά χρόνια, σε production περιβάλλοντα (πχ Netflix). Η διαχείρισή της όμως, παραμένει δύσκολη και απαιτεί προσεκτικές κινήσεις και καλή γνώση του διαχειριστή. Έχουν υπάρξει προσπάθειες αυτοματοποίησης της διαχείρισης της Cassandra, αλλά μέχρι τώρα είναι είτε κλειστό εμπορικό λογισμικό, είτε δουλεύουν μόνο σε συγκεκριμένους Cloud Providers.

Από την άλλη, ο Kubernetes είναι μία πλατφόρμα διαχείρισης εργασιών σε Containers, που τρέχει σε οποιοδήποτε περιβάλλον (Cloud, On-Prem) και προσφέρει αρκετή έτοιμη λειτουργικότητα για διαχείριση εφαρμογών αποθήκευσης δεδομένων.

Θέλουμε λοιπόν να φτιάξουμε ένα λογισμικό διαχείρισης της Apache Cassandra, το οποίο θα είναι ανοιχτού κώδικα και θα μπορεί να λειτουργήσει σε οποιοδήποτε περιβάλλον. Για τον λόγο αυτό, θα προσπαθήσουμε να αναπτύξουμε το λογισμικό διαχείρισης της Cassandra πάνω στην πλατφόρμα Kubernetes. Θα αξιολογήσουμε τις έτοιμες λύσεις που προσφέρει ο Kubernetes για τη διαχείριση λογισμικού αποθήκευσης δεδομένων, θα δούμε γιατί δεν μπορούν να χρησιμοποιηθούν όπως είναι για την Cassandra και θα τις επεκτείνουμε ώστε να τρέξουμε μια λειτουργική Cassandra με πλήρως αυτοματοποιημένη διαχείριση.

0.1.2 Υπάρχουσες Προσεγγίσεις

Οι υπάρχουσες προσεγγίσεις χωρίζονται σε δύο κατηγορίες: σε αυτές που χρησιμοποιούν την πλατφόρμα του Kubernetes και σε αυτές που δεν την χρησιμοποιούν.

Το Netflix Priam είναι μια προσέγγιση από τη Netflix, για αυτόματη διαχείριση της Apache Cassandra, που όμως λειτουργεί μόνο στην πλατφόρμα Amazon Web Services, αναγκάζοντας όσους θέλουν να το χρησιμοποιήσουν να είναι πελάτες της Amazon.

Στον Kubernetes, οι επίσημες οδηγίες συμβουλεύουν να τη χρήση του μηχανισμού StatefulSet για εφαρμογές αποθήκευσης δεδομένων όπως η Cassandra. Όμως, όπως θα αναλύσουμε, ο μηχανισμός αυτός δεν είναι επαρκής και πρέπει να τον επεκτείνουμε.

Τέλος, οι πιο κοντινές στη δική μας λύσεις, αφορούν επέκταση του Kubernetes. Τέτοιες προσπάθειες είναι ο Cassandra Operator από Instaclustr και ο Navigator από Jetstack. Σε σχέση με αυτές τις λύσεις, διαφοροποιούμαστε στον σχεδιασμό μας, ο οποίος πιστεύουμε πως εξασφαλίζει καλύτερη και ευκολότερη διαχείριση της Cassandra.

0.2 Υπόβαθρο

Στην ενότητα αυτή θα παρουσιάσουμε τα βασικά θεωρητικά στοιχεία που είναι απαραίτητα για την κατανόηση της εργασίας μας, δίνοντας έμφαση στην αρχιτεκτονική και στη λειτουργία του Kubernetes και της Cassandra.

0.2.1 Apache Cassandra

Η Apache Cassandra είναι μια βάση δεδομένων, παρόμοια με μια SQL βάση δεδομένων. Αυτό σημαίνει ότι χρήστες γράφουν και διαβάζουν δεδομένα από την Cassandra, σε μορφή γραμμών πίνακα.

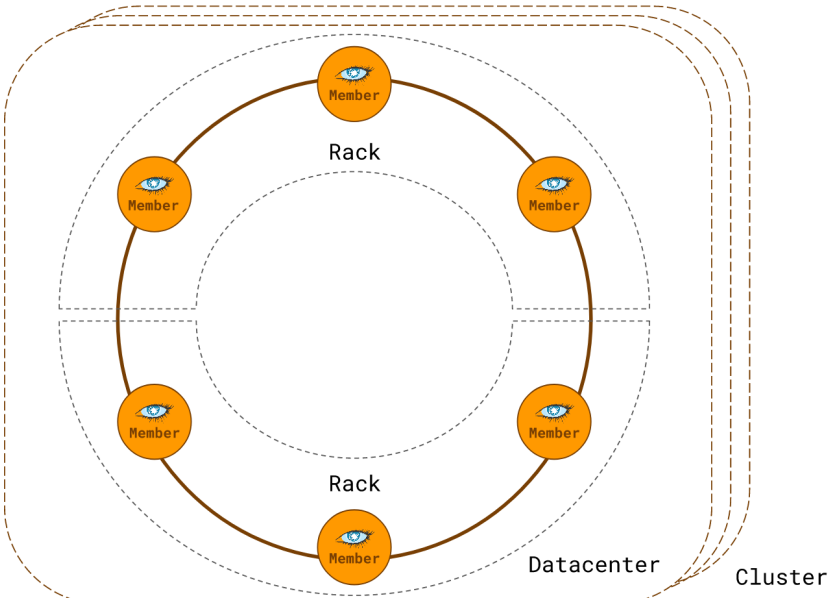
0.2.1.1 Κατανεμημένη Αρχιτεκτονική

Η Cassandra είναι μια κατανεμημένη βάση δεδομένων. Οι έννοιες που χρησιμοποιεί η Cassandra για να περιγράψει την τοπολογία ενός Cassandra Cluster είναι:

- **Μέλος**: ο βασικός δομικός λίθος ενός Cassandra Cluster. Ένας Μέλος είναι συνήθως ένας υπολογιστής.
- **Rack**: Πολλά Μέλη σχηματίζουν ένα Rack. Το Rack συνήθως αντιστοιχίζεται σε ένα Availability Zone (πχ Αθήνα, Βερολίνο, eu-west1-b).
- **Datacenter**: Πολλά Racks σχηματίζουν ένα Datacenter. Το Datacenter συνήθως αντιστοιχίζεται σε ένα Region (πχ Ελλάδα, Γερμανία, eu-west1).
- **Cluster**: Πολλά Datacenters σχηματίζουν ένα Cassandra Cluster. Το Cluster είναι παγκόσμιο, αφού αποτελείται από Datacenters που βρίσκονται σε πολλά σημεία του κόσμου.

0.2.1.2 Κατανομή των Δεδομένων

Η Cassandra, όπως αναφέραμε, χρησιμοποιεί μια κατανεμημένη αρχιτεκτονική. Τρέχει σε πολλά μέρη ταυτόχρονα και έτσι αντέχει στις διάφορες υλικές καταστροφές



Σχήμα 1

που μπορούν να συμβούν. Μια ερώτηση που προκύπτει εύκολα είναι η εξής: πώς μοιράζονται τα δεδομένα ανάμεσα στα μέλη ενός Cassandra Cluster?

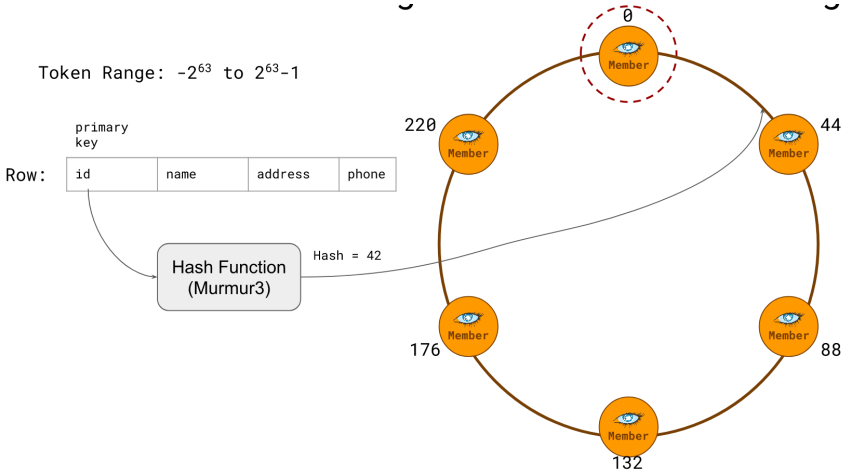
Θέλουμε τα δεδομένα να είναι μοιρασμένα, καθώς δεν γίνεται όλα τα μέλη να έχουν όλα τα δεδομένα. Αυτή η διαδικασία πρέπει να είναι:

- Δίκαιη, δηλαδή κάθε μέλος να έχει περίπου ίδιο όγκο δεδομένων.
- Σταθερή, δηλαδή η καταστροφή ενός Μέλους να οδηγεί μόνο στην ανακατανομή των δεδομένων του συγκεκριμένου Μέλους.
- Αυτόματη, δηλαδή τα δεδομένα θα κατανέμονται χωρίς τη συμβολή του χρήστη, σε αντίθεση με μηχανισμούς sharding που υπάρχουν σε άλλες βάσεις δεδομένων.

Αυτό είναι ένα πρόβλημα που έχει μελετηθεί εκτεταμένα στη βιβλιογραφία και η Cassandra επιλέγει να το λύσει με μία τεχνική που ονομάζεται Consistent Hashing και λειτουργεί ως εξής:

- Ένα εύρος αριθμών, το $[-2^{63}, 2^{63})$, απεικονίζεται πάνω σε ένα δαχτυλίδι.
- Κάθε μέλος παίρνει έναν αριθμό πάνω σε αυτό το δαχτυλίδι. Αυτοί ο αριθμός ονομάζεται token του Μέλους (κάθε μέλος έχει παραπάνω από 1 tokens, αλλά εδώ τα απεικονίζουμε με 1 για λόγους απλότητας).

- Όταν πρέπει να προσπελάσουμε μια γραμμή, παίρνουμε το πρωτεύον κλειδί της γραμμής και το περνάμε από μία συνάρτηση κατακερματισμού (συνήθως Murmur3), η οποία μας δίνει έναν αριθμό πάνω στο δαχτυλίδι.
- Γνωρίζοντας τα tokens των Μελών και τον αριθμό της γραμμής, βρίσκουμε σε ποιο Μέλος ανήκει.



Σχήμα 2

0.2.1.3 Μεγαλώνοντας ένα Cassandra Cluster

Αφού είδαμε μερικά βασικά χαρακτηριστικά της Cassandra, θα δείξουμε πώς πρέπει να γίνονται συγκεκριμένες διαχειριστικές ενέργειες. Γνωρίζοντας τον σωστό τρόπο, θα δούμε γιατί τα StatefulSets τις εκτελούν λανθασμένα και θα επεκτείνουμε τον Kubernetes ώστε να χειρίζεται σωστά τα Cassandra Clusters.

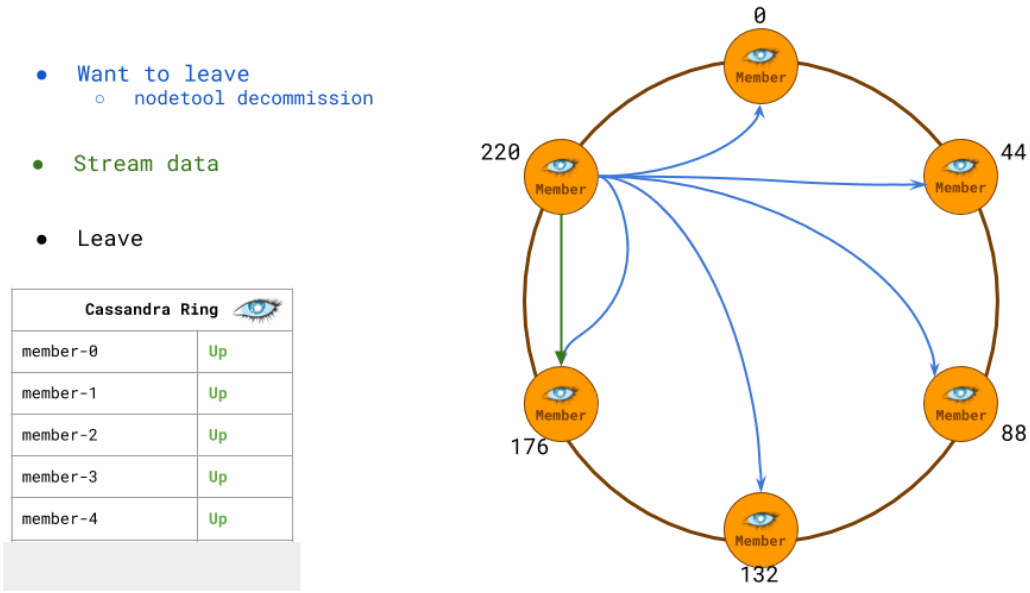
Η διαδικασία για να μεγαλώσω ένα Cassandra Cluster είναι η εξής:

1. Ο διαχειριστής θέλει να βάλει ένα νέο Μέλος. Επικοινωνεί με κάποια Μέλη του Cassandra Cluster, που λέγονται Seeds και του τα δίνει ο διαχειριστής, για να το βάλουν μέσα. Είναι λοιπόν σημαντικό, να υπάρχουν πάντα διαθέσιμα Μέλη που να είναι Seeds.
2. Τα Seeds επικοινωνούν με τα υπόλοιπα μέλη και αποφασίζουν ποια tokens, δηλαδή ποιο μέρος του δαχτυλιδιού, θα δώσουν στο νέο Μέλος.
3. Το νέο Μέλος παίρνει τα δεδομένα που του αντιστοιχούν από τα υπόλοιπα Μέλη και είναι πλέον κανονικό Μέλος του Cassandra Cluster.

0.2.1.4 Μικραίνοντας ένα Cassandra Cluster

Η διαδικασία για να μικρύνω ένα Cassandra Cluster είναι η εξής:

1. Ο διαχειριστής θέλει να βγάλει ένα Μέλος. Του δίνει την εντολή να βγει (`nodetool decommission`).
2. Το Μέλος λέει στα άλλα μέλη ότι θέλει να φύγει. Γίνεται ανακατανομή των tokens και τα δεδομένα στέλνονται στα Μέλη που είναι τώρα υπεύθυνα για αυτά.
3. Το Μέλος βγαίνει από το Cassandra Cluster.



Σχήμα 3

0.2.2 Kubernetes

0.2.2.1 Containers

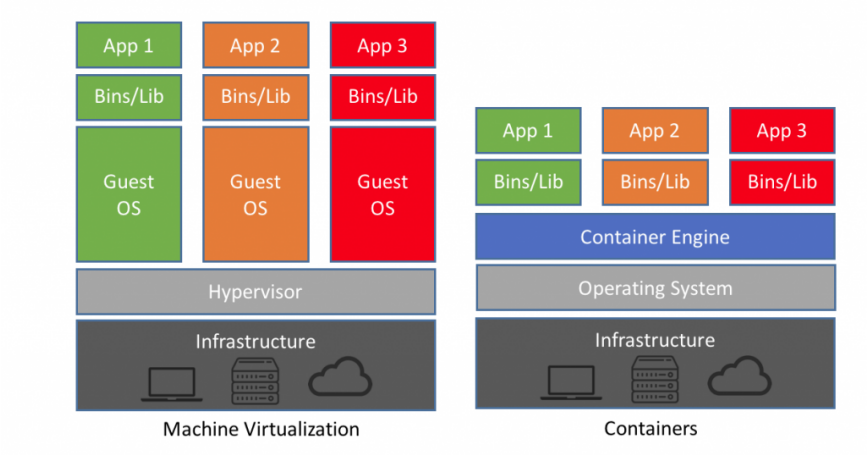
Όπως αναφέραμε προηγουμένως, ο Kubernetes είναι μια πλατφόρμα διαχείρισης εργασιών που τρέχουν σε Containers. Τι είναι όμως ένα Container?

Ακριβώς όπως οι Εικονικές Μηχανές (Virtual Machines) προσφέρουν ένα απομονωμένο περιβάλλον για να τρέξει μια εφαρμογή, το ίδιο πετυχαίνουν και τα Containers.

Χρησιμοποιώντας μηχανισμούς του Linux πηρύνα και συγκεκριμένα τους namespaces και cgroups, τα Containers δημιουργούν ένα απομονωμένο και ελεγχόμενο περιβάλλον που μπορεί να τρέξει μια διεργασία.

Η διεργασία που τρέχει μέσα σε ένα Container βλέπει μόνο διεργασίες, αντάπτορες δικτύου και σύστημα αρχείων που αντιστοιχούν στο Container της, όπως ακριβώς σε μία Εικονική Μηχανή. Επίσης οι πόροι της, όπως επεργαστική ισχύς και μέγεθος μνήμης, είναι καθορισμένοι.

Άρα τα Containers δημιουργούν ένα απομονωμένο και ελεγχόμενο περιβάλλον, όπως ακριβώς θα έκανε μια Εικονοκή Μηχανή, χωρίς όμως τον επιπλέον υπολογιστικό φόρτο που επιφέρει η διαδικασία προσομοίωσης ενός Guest Λειτουργικού Συστήματος. Αυτό συμβαίνει επειδή τα Containers χρησιμοποιούν τον πηρύνα του Host Λειτουργικού Συστήματος.



Σχήμα 4

Το λογισμικό Docker είναι αυτό που έκανε διάσημο τον μηχανισμό των Containers και πρόσθεσε μία πολύ σημαντική λειτουργία, τα Images. Ένα Image είναι ένα πακετάρισμα ενός συνόλου αρχείων, που είναι immutable και έτσι μπορεί να αναπαραχθεί από όποιον έχει το Image. Έτσι, οι εφαρμογές πακετάρονται μαζί με τις βιβλιοθήκες που χρειάζονται σε ένα Image, το οποίο παράγει ένα περιβάλλον για να τρέξει η εφαρμογή. Το περιβάλλον αυτό είναι ίδιο ανεξάρτητα με το πού τρέχει η εφαρμογή, λύνοντας έτσι το πρόβλημα του reproducibility και επιτρέποντας στους developers να έχουν το ίδιο περιβάλλον σε dev, staging και production.

Συνοψίζοντας λοιπόν, η ιδέα είναι ότι έχω Images που περιέχουν μία εφαρμογή με τις βιβλιοθήκες της και την τρέχω σε ένα απομονωμένο, ελεγχόμενο και reproducible

περιβάλλον.

0.2.2.2 Αντικείμενα του Kubernetes

Ο Kubernetes αντιπροσωπεύει τις εργασίες και τους πόρους του συστήματος χρησιμοποιώντας αντικείμενα. Ένα Kubernetes αντικείμενο είναι απλώς ένα κομμάτι κειμένου γραμμένο στη γλώσσα YAML. Ένα Kubernetes αντικείμενο έχει δύο μέρη:

- Το κομμάτι **Spec**, το οποίο γράφει ο χρήστης και αντιπροσωπεύει την επιθυμητή κατάσταση του συστήματος. Για παράδειγμα, έστω ότι ο χρήστης θέλει να τρέξει ένα Web Server με την εφαρμογή Nginx. Το αντικείμενο του Kubernetes που τρέχει Containers ονομάζεται Pod. Έτσι ο χρήστης γράφει το παρακάτω αντικείμενο:

Pod Example

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: web-server
spec:
  containers:
    - name: nginx
      image: nginx:1.15
      ports:
        containerPort: 80
```

  Ο Kubernetes αποθηκεύει αυτό το αντικείμενο και προσπαθεί συνεχώς ώστε να εκπληρώσει την επιθυμία του χρήστη.

- Το κομμάτι **Status**, το οποίο ανανεώνει ο Kubernetes και εκφράζει την πραγματική κατάσταση του συστήματος. Στο παράδειγμα του Pod, το πεδίο Status θα περιέχει πληροφορία σχετικά με το αν το Pod τρέχει, αν έχει δρομολογηθεί, κλπ.

0.2.2.3 Το Πρότυπο Σχεδίασης Controller

Αναφέραμε ότι ο Kubernetes αντιπροσωπεύει τις εργασίες και τους πόρους του συστήματος χρησιμοποιώντας αντικείμενα. Ο χρήστης δηλώνει την επιθυμητή κατάσταση του αντικειμένου (Spec) και ο Kubernetes κάνει ό,τι μπορεί για να την πραγματοποιήσει. Πώς το καταφέρνει όμως αυτό?

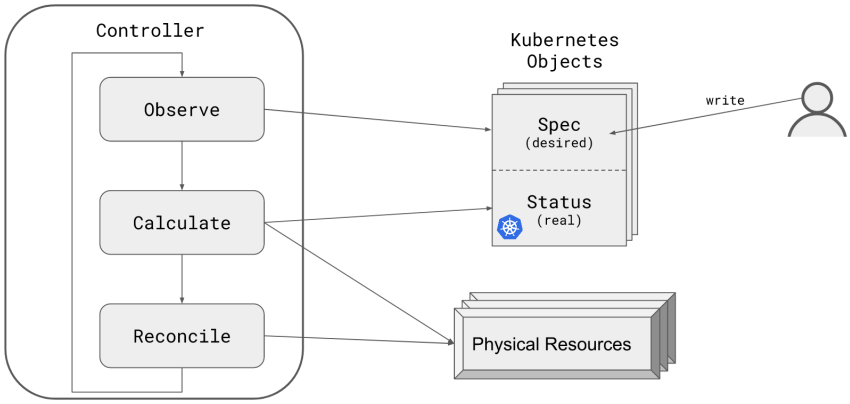
Όλη η λογική στον Kubernetes υλοποιείται από κάποιες διεργασίες που λέγονται Controllers. Ένα παράδειγμα Controller είναι το kubelet, το πρόγραμμα που τρέχει σε κάθε κόμβο του Kubernetes Cluster και είναι υπεύθυνο για να διαχειρίζεται τα Containers που πρέπει να τρέξουν στον συγκεκριμένο κόμβο.

Οι Controllers του Kubernetes εκτελούν συνεχώς την ακόλουθη επανάληψη:

- **Παρατηρούν** την επιθυμητή κατάσταση του χρήστη (Spec) για τον τύπο αντικειμένων που διαχειρίζονται. Για παράδειγμα, το kubelet παρατηρεί το Spec του web-server Pod και βλέπει ότι πρέπει να τρέξει ένα container με το image του Nginx.
- **Υπολογίζουν** την πραγματική κατάσταση του συστήματος (Status) μιλώντας με φυσικούς πόρους. Για παράδειγμα, το kubelet επικοινωνεί με το Docker runtime στον κόμβο του και βλέπει ότι το container για το Pod web-server δεν τρέχει ακόμα.
- **Εκτελούν ενέργειες** ώστε η πραγματική κατάσταση να έρθει πιο κοντά στην επιθυμητή. Για παράδειγμα, το kubelet επικοινωνεί με το Docker runtime και δημιουργεί ένα νέο container που τρέχει το image του Nginx.

0.2.2.4 StatefulSet

Το Pod είναι ένα πολύ χρήσιμο αντικείμενο του Kubernetes, που χρησιμοποιείται ως δομικός λίθος για να τρέξουμε διάφορες εφαρμογές. Μόνο του, είναι δύσκολο να χρησιμοποιηθεί για να τρέξει μια εφαρμογή αποθήκευσης δεδομένων. Χρειάζεται κάποιος να συντονίζει τη δημιουργία των Pods, να τους δίνει αποθηκευτικό χώρο καθώς και μια σταθερή δικτυακή ταυτότητα. Η λύση του Kubernetes σε αυτή την περίπτωση είναι το StatefulSet.



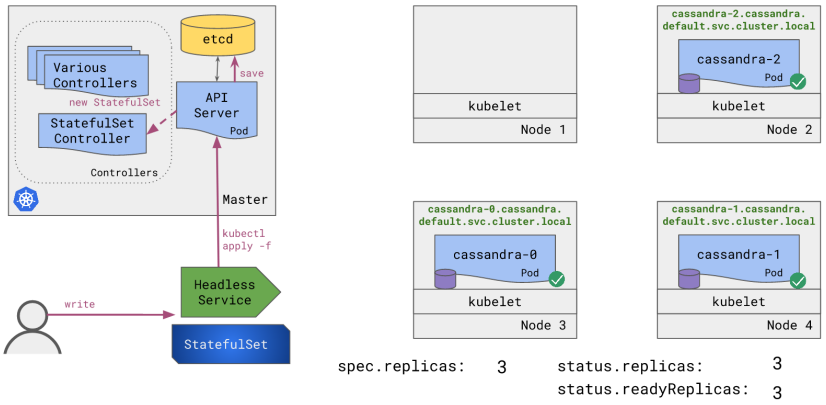
Σχήμα 5

Το StatefulSet είναι ουσιαστικά ένα σύνολο από Pods, τα οποία ο Kubernetes χειρίζεται με συγκεκριμένο τρόπο. Το StatefulSet είναι η πρόταση του Kubernetes σε όποιον θέλει να τρέξει μια εφαρμογή αποθήκευσης δεδομένων, όπως είναι η Apache Cassandra. Για την ακρίβεια, στον επίσημο οδηγό χρήσης του Kubernetes, υπάρχει παράδειγμα χρήσης του StatefulSet για την δημιουργία ενός Cassandra Cluster. Συνοπτικά, το StatefulSet δίνει στον χρήστη τις εξής εγγυήσεις:

- **Δίσκος**: Κάθε Pod ενός StatefulSet παίρνει τον δικό του δίσκο για να αποθηκεύει τα δεδομένα του.
- **Δικτυακή Ταυτότητα**: Κάθε Pod ενός StatefulSet έχει τη δική του δικτυακή ταυτότητα υπό τη μορφή ενός DNS record. Για παράδειγμα, το DNS Record ”member-0.default.svc.cluster.local” θα δείχνει πάντα στο Pod ”member-0”.
- **Προσεκτική Διαχείριση**: Το StatefulSet είναι προσεκτικό στο πώς διαχειρίζεται τα Pods του. Όταν δημιουργείται ένα StatefulSet, τα Pods του δεν δημιουργούνται όλα μαζί, αλλά ένα προς ένα και αφού όλα τα προηγούμενα είναι υγιή και έτοιμα. Το ίδιο και όταν ανανεώνεται το Spec ενός StatefulSet, οι αλλαγές εφαρμόζονται σε ένα Pod τη φορά, σε μια διαδικασία που λέγεται Rolling Update.

0.2.2.5 Ελλείψεις του StatefulSet

Ο Kubernetes προτείνει να τρέξουμε την Cassandra σε ένα StatefulSet. Μάλιστα, ο επίσημος οδηγός του Kubernetes δίνει παράδειγμα του να τρέχεις την Cassandra



Σχήμα 6

σε ένα StatefulSet. Ενώ το StatefulSet πράγματι προσφέρει κάποια πολύ χρήσιμη λειτουργικότητα, δεν είναι αρκετό ώστε να διαχειρίζεται σωστά την Cassandra.

Θα παρουσιάσουμε 3 βασικά προβλήματα του StatefulSet:

- **Αδυναμία Γεωγραφικής Κατανομής**: Όπως αναφέραμε, η Cassandra κατανέμεται γεωγραφικά ώστε να αντέχει σε υλικές καταστροφές. Όμως με το να τρέχουμε την Cassandra σε 1 StatefulSet, την περιορίζουμε σε 1 Availability Zone. Αυτό συμβαίνει επειδή μπορούμε να κλειδώσουμε τα Pods του StatefulSet σε κάποιο συγκεκριμένο topology domain, δεν μπορούμε να πούμε δηλαδή θέλω 2 Pod στην Αθήνα και 2 στην Θεσσαλονίκη.

- **Λάθος Διαδικασία Σμίκρυνσης**: Στην προηγούμενη υποενότητα, περιγράψαμε τη διαδικασία που πρέπει να ακολουθήσει ένας διαχειριστής για να μικρύνει ένα Cassandra Cluster. Χρειάζεται να δώσει εντολή στο Μέλος να φύγει, να περιμένει μέχρι το Μέλος να έχει στείλει όλα του τα δεδομένα στα άλλα Μέλη και μετά να το αφαιρέσει. Το StatefulSet δεν ακολουθεί αυτή τη διαδικασία και δεν μας δίνει τη δυνατότητα να εισάγουμε αυτή τη λειτουργικότητα, ίσως με τη μορφή κάποιου hook (σημείου που καλεί τον κώδικά μας). Με άλλα λόγια, το StatefulSet δεν λέει στη διεργασία αν φεύγει για πάντα, απλά ότι πρέπει να σταματήσει, το οποίο μπορεί να συμβεί για πολλούς άλλου λόγους.

- **Δικτυακή Ταυτότητα με DNS Records**: Η Cassandra εσωτερικά αποθηκεύει τα Μέλη της με την διεύθυνση IP τους και ένα μοναδικό αλφαριθμητικό UUID (host_id) το οποίο παράγει όταν το Μέλος μπαίνει στο Cassandra Cluster.

Όμως στον Kubernetes δεν έχουμε σταθερές IP διευθύνσεις, αλλά μόνο σταθερά DNS Records. Αυτό σημαίνει ότι δεν γνωρίζουμε την ταυτότητα του Μέλους πριν το εισάγουμε, το οποίο είναι πρόβλημα αν συμβεί κάποια βλάβη και το μέλος μετά πρέπει να βρει αν είναι ήδη Μέλος του Cassandra Cluster ή όχι. Γενικά, θα προτιμούσαμε να έχουμε σταθερές IP διευθύνσεις καθώς θα απλοποιούσε πολύ τις διαδικασίες.

Εκτός από αυτά υπάρχουν και αρκετά άλλα στα οποία δεν θα μπούμε σε λεπτομέρεια:

- Backups & Restores
- Multi-Region Cassandra Clusters
- Έλλειψη Μελλοντικής Επεκτασιμότητας

0.3 Σχεδίαση

Στο κεφάλαιο αυτό, θα σχεδιάσουμε μια λύση που θα διαχειρίζεται αυτόματα την Apache Cassandra στον Kubernetes. Στο προηγούμενο κεφάλαιο, μελετήσαμε τις ελλείψεις του StatefulSet και τους λόγους που δεν μπορεί από μόνο του να αποτελέσει τη λύση που ψάχνουμε.

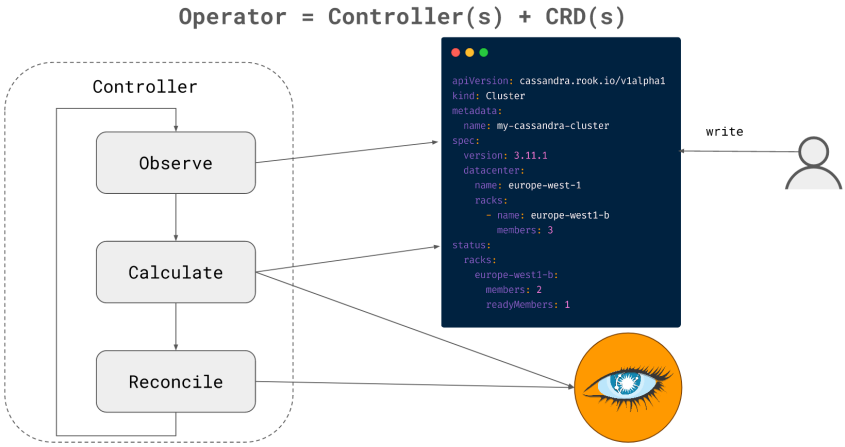
0.3.1 Operator Pattern

Στην αναζήτησή μας για λύση, ανατρέχουμε στις σχεδιαστικές αποφάσεις του Kubernetes. Στον Kubernetes τα πάντα αντιπροσωπεύονται με αντικείμενα, στα οποία ο χρήστης γράφει την επιθυμητή κατάσταση, ο Kubernetes τη διαβάζει και υπολογίζει την πραγματική κατάσταση και εκτελεί ενέργειες ώστε να φέρει την πραγματική κατάσταση πιο κοντά στην επιθυμητή. Το πρότυπο αυτό ονομάζεται **Controller Pattern** και χρησιμοποιείται παντού στον Kubernetes. Τα προγράμματα που υλοποιούν την επανάληψη που περιγράψαμε λέγονται Controllers.

Θα μπορούσαμε λοιπόν να ακολουθήσουμε το πρότυπο σχεδίασης του Kubernetes? Δηλαδή, να αναπαραστήσουμε το Cassandra Cluster ως ένα Αντικείμενο του Kubernetes.

Ο χρήστης θα συμπληρώνει την επιθυμητή κατάσταση (Spec) του Cassandra Cluster και ένας δικός μας Controller θα υπολογίζει την πραγματική κατάσταση, θα την συγκρίνει με την πραγματική και θα εκτελεί ενέργειες ώστε να φέρει την πραγματική κατάσταση πιο κοντά στην επιθυμητή.

Αυτή η προσέγγιση ονομάζεται **Operator Pattern**, επειδή ενσωματώνει την γνώση του ανθρώπου διαχειριστή σε ένα πρόγραμμα που διαχειρίζεται σωστά την εφαρμογή αποθήκευσης δεδομένων.



Σχήμα 7

Παραμένει όμως το πρόβλημα της αναπαράστασης του Cassandra Cluster ως ένα Αντικείμενο του Kubernetes. Πώς θα πούμε στον Kubernetes να αποθηκεύσει τα δικά μας Cassandra Cluster Αντικείμενα, για τα οποία είναι υπεύθυνος ο δικός μας Controller; Η απάντηση βρίσκεται στον μηχανισμό επέκτασης Custom Resource Definition (CRD).

Το CRD είναι ένα Αντικείμενο του Kubernetes, το οποίο αντιπροσωπεύει τη δήλωση ενός δικού μας, custom Αντικειμένου. Ο Kubernetes δημιουργεί REST HTTP Paths για να διαβάζουμε και να αποθηκεύουμε τα custom Αντικείμενά μας. Για παράδειγμα, το CRD για τα Cassandra Cluster αντικείμενα είναι το εξής:

Cassandra Cluster CRD

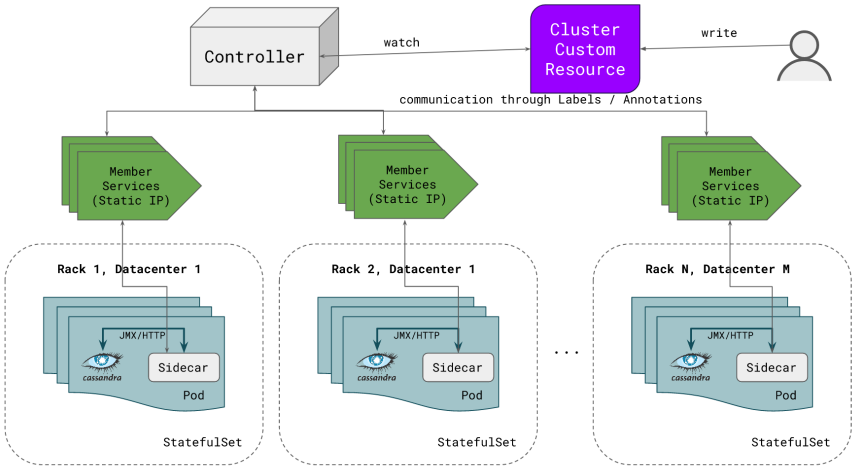
```
# Cassandra Cluster CRD
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
```

```
  name: clusters.cassandra.rook.io
spec:
  group: cassandra.rook.io
  names:
    kind: Cluster
    listKind: ClusterList
    plural: clusters
    singular: cluster
  scope: Namespaced
version: v1alpha1
```

0.3.2 Βασικές Σχεδιαστικές Αποφάσεις

Για τους λόγους που αναφέραμε στο προηγούμενο κεφάλαιο, η απλή χρήση StatefulSet δεν μας εξυπηρετεί πλήρως. Έτσι, επεκτείνουμε τον Kubernetes χρησιμοποιώντας το Operator Pattern, για να φτιάξουμε λογισμικό αυτόματης διαχείρισης της Cassandra.

Μια συνοπτική παρουσίαση της σχεδίασής μας φαίνεται στην παρακάτω εικόνα:



Σχήμα 8

Οι βασικές σχεδιαστικές αποφάσεις που πήραμε είναι:

- Χρήση του Operator Pattern για επέκταση του Kubernetes.

- Επαναχρησιμοποίηση του StatefulSet καθώς προσφέρει αρκετή λειτουργικότητα που χρειαζόμαστε. Δεν θα έχουμε 1 StatefulSet για κάθε Cassandra Cluster, αλλά 1 StatefulSet για κάθε Rack του κάθε Datacenter.
- Στατικές διευθύνσεις IP, τις οποίες υλοποιούμε με τα Αντικείμενα τύπου Service του Kubernetes.
- Ένα μικρό δικό μας πρόγραμμα (sidecar) που θα τρέχει μαζί με τη διεργασία της Cassandra σε κάθε Μέλος και θα μας βοηθάει σε λειτουργίες που απαιτούν πρόσβαση στο σύστημα αρχείων.

Θα αναλύσουμε καθεμιά από αυτές τις αποφάσεις.

0.3.2.1 Χρήση του Operator Pattern

Operator Pattern ονομάζεται το πρότυπο σχεδίασης στο οποίο το λογισμικό αποθήκευσης δεδομένων αναπαρίσταται με ένα Αντικείμενο του Kubernetes, το οποίο δηλώνουμε στον Kubernetes με χρήση Custom Resource Definitions, σε συνδυασμό με έναν Controller που κωδικοποιεί τη γνώση του ανθρώπου διαχειριστή για το πώς πρέπει να διαχειρίζεται σωστά το λογισμικό αυτό.

Το Cassandra Cluster Αντικείμενο που γράφει ο χρήστης για να ζητήσει ένα Cassandra Cluster, είναι το παρακάτω:

Παράδειγμα Αντικειμένου Τύπου Cassandra Cluster

```
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  repository: my-private-repo.io/cassandra
  mode: cassandra
  datacenter:
    name: us-east-1
```

```
racks:
  - name: us-east-1a
    members: 3
    storage:
      volumeClaimTemplates:
        - metadata:
            name: rook-cassandra-data
          spec:
            storageClassName: my-storage-class
            resources:
              requests:
                storage: 200Gi
    resources:
      requests:
        cpu: 8
        memory: 32Gi
      limits:
        cpu: 8
        memory: 32Gi
    placement:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/region
                operator: In
                values:
                  - us-east-1
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - us-east-1a
```

0.3.2.2 Επαναχρησιμοποίηση του StatefulSet

Το StatefulSet Αντικείμενο δεν μας ικανοποιεί όπως είναι, παρόλα αυτά όμως περιέχει αρκετή λειτουργικότητα που χρειαζόμαστε. Για τον λόγο αυτό, θα το χρησιμοποιήσουμε σαν δομικό λίθο στην προσέγγισή μας. Η αντιστοίχιση εννοιών που κάνουμε από την Cassandra στον Kubernetes είναι η εξής:

Cassandra	Kubernetes
Member	Pod
Rack	StatefulSet
Datacenter	Pool of StatefulSets
Cluster	Cluster CRD

Δηλαδή πλέον δημιουργούμε 1 StatefulSet ανά Rack για κάθε Datacenter. Αυτό μας επιτρέπει να κατανείμουμε τα μέλη της Cassandra γεωγραφικά, κάτι που δεν μας επέτρεπε το απλό StatefulSet.

0.3.2.3 Στατικές IP

Όπως αναφέραμε σε προηγούμενο κεφάλαιο, η χρήση καταχωρήσεων DNS μας δημιουργεί προβλήματα και δυσκολεύει την διαχείριση της Cassandra, αφού η Cassandra αποθηκεύει τα Μέλη με την IP διεύθυνσή τους και η διεύθυνση αυτή αλλάζει συχνά στον Kubernetes.

Τα πράγματα θα απλοποιούνταν αρκετά, αν καταφέρναμε να εξασφαλίσουμε στατικές IP διευθύνσεις για τα Μέλη του Cassandra Cluster. Στην αναζήτησή μας, παρατηρήσαμε ότι ένα Αντικείμενο του Kubernetes μπορεί να μας εξασφαλίσει στατικές IP. Αυτό το Αντικείμενο λέγεται Service.

Τα Services παρακολουθούν ένα σύνολο από Pods, το οποίο επιλέγεται μέσω κάποιων ζευγαριών key-value που λέγονται Labels και αποτελούν μεταδεδομένα πάνω σε Αντικείμενα του Kubernetes. Παράλληλα, παράγουν μια IP διεύθυνση, που ονομάζεται ClusterIP, η οποία δεν αλλάζει. Οποιαδήποτε κλήση έρχεται σε αυτή τη διεύθυνση, την μεταβιβάζουν σε ένα από τα backend Pods.

Κανονικά τα Services προορίζονται για χρήση από ένα σύνολο Pods. Για να εξυπηρετήσουμε την περίπτωσή μας, θα δημιουργούμε ένα ClusterIP Service ανά Pod,

δηλαδή το κάθε μέλος της Cassandra θα έχει τη δικιά του στατική IP, την ClusterIP του Service.

Η προσέγγιση αυτή μελετήθηκε ως προς τα μειονεκτήματα της, αλλά δεν βρέθηκαν ανατρεπτικοί λόγοι. Πιθανά μειονεκτήματα μια τέτοιας προσέγγισης είναι:

- **Μειωμένη Απόδοση**: Τα ClusterIP Services υλοποιούνται με κανόνες iptables και δεν παρουσιάζουν σημαντική μείωση στην απόδοση του δικτύου μέχρι μερικές εκατοντάδες. Για μεγαλύτερους αριθμούς, υπάρχει υλοποίηση με IPVS η οποία είναι σταθερή και μπορεί να χρησιμοποιηθεί σε production περιβάλλοντα.
- **Εξάντληση IP Διευθύνσεων**: Κάθε Kubernetes Cluster τυπικά διαθέτει ένα /12 IP Block, δηλαδή $2^{12} = 4096$ IP διευθύνσεις για να δώσει σε ClusterIP Services. Συνεπώς, η εξάντληση των IP διευθύνσεων δεν θα αποτελέσει πρόβλημα.

0.3.2.4 Sidecar

Αρκετές ενέργειες απαιτούν πρόσβαση στο σύστημα αρχείων του Μέλους. Τέτοιες ενέργειες αποτελούν:

- Δημιουργία και επεξεργασία των ρυθμίσεων της Cassandra που βρίσκονται μέσα σε αρχεία κειμένου.
- Εγκατάσταση plugins.
- Backups και Restores.

Για τον λόγο αυτό επιλέξαμε να περιλάβουμε μια μικρή διεργασία που θα τρέχει μαζί με την Cassandra και θα βοηθάει σε αυτές τις ενέργειες. Η διεργασία αυτή ονομάζεται sidecar, γιατί τρέχει μαζί με την κύρια διεργασία και την βοηθάει.

0.3.3 Σχεδιασμός Ενεργειών

Αφού καθορίσαμε τους βασικούς πυλώνες του σχεδιασμού μας, ήρθε η ώρα να περιγράψουμε τις ακολουθίες ενεργειών που θα γίνονται ώστε να εξασφαλίσουν τη σωστή δημιουργία και λειτουργία ενός Cassandra Cluster.

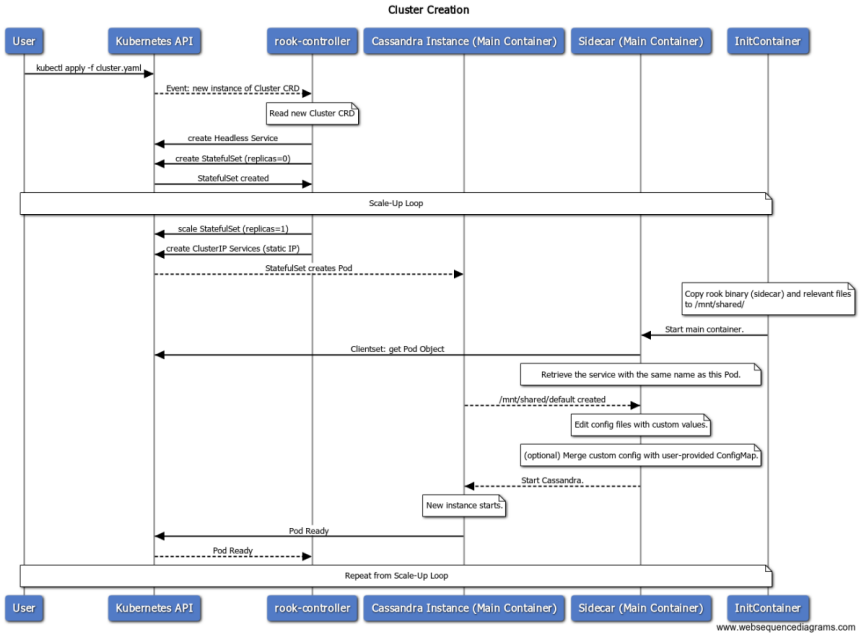
Θα εξετάσουμε τα εξής:

- Δημιουργία και μεγένθυση ενός Cassandra Cluster
- Σμίκρυνση ενός Cassandra Cluster
- Χρήση Τοπικών Δίσκων

0.3.3.1 Δημιουργία και μεγένθυση ενός Cassandra Cluster

Πρώτα από όλα, θα σχεδιάσουμε την ακολουθία ενεργειών που χρειάζονται για την δημιουργία και τη μεγέθυνση ενός Cassandra Cluster. Τη δημιουργία ενός Cassandra Rack με n Μέλη, μπορούμε να τη σκεφτούμε ως τη δημιουργία ενός Rack με 0 Μέλη και τη μεγένθυσή του, ένα προς ένα, σε n Μέλη.

Με βάση αυτό, η ακολουθία ενεργειών που ακολουθεί ο Cassandra Operator φαίνεται στο παρακάτω διάγραμμα:



Σχήμα 9

Επεξήγηση:

1. **Ο Χρήστης** δημιουργεί ένα Cassandra Cluster Αντικείμενο, δηλώνοντας την επιθυμητή κατάσταση για το Cassandra Cluster του.

```
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: us-east-1
    racks:
      - name: us-east-1a
        members: 3
        storage:
          volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: my-storage-class
                resources:
                  requests:
                    storage: 200Gi
        resources:
          requests:
            cpu: 8
            memory: 32Gi
          limits:
            cpu: 8
            memory: 32Gi
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
```

```
          nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - us-east-1a
```

2. **Ο Controller** ενημερώνεται ότι υπάρχει ένα καινούριο Cluster Αντικείμενο. Για κάθε Rack σε κάθε Datacenter, δημιουργεί:

   (a) i. Αν δεν υπάρχει, ένα StatefulSet με 0 Μέλη για το Rack.

   ii. Αν υπάρχει το StatefulSet, ελέγχει αν ο επιθυμητός αριθμός Μελών του είναι ίσος με τον πραγματικο. Αν χρειάζεται περισσότερα μέλη και όλα τα υπόλοιπα Μέλη του Cluster είναι υγιή και έτοιμα, αυξάνει τον αριθμό Pods του StatefulSet κατά 1.

   (b) Ένα ClusterIP Service για κάθε Μέλος του Cassandra Cluster. Τα Services αυτά έχουν παρέχουν μια στατική IP διεύθυνση και έχουν ίδιο όνομα με το Μέλος στο οποίο αντιστοιχούν.

   Ο Controller επίσης δημιουργεί ένα Headless Service, το οποίο δημιουργεί μια καταχώρηση DNS στην οποία μπορούν να συνδέονται χρήστες για να κατευθύνονται σε ένα υγιές και έτοιμο Μέλος του Cassandra Cluster.

3. **Το Μέλος** (StatefulSet Pod) ξεκινά και τρέχει ένα init container:

   (a) Το init container ξεκινά και αντιγράφει το sidecar binary και άλλα απαραίτητα αρχεία (όπως plugins) σε ένα μοιραζόμενο φάκελο (/mnt/shared).

   (b) Το Container που θα τρέξει τη διεργασία της Cassandra ξεκινά, τρέχοντας το δικό μας sidecar binary. entrypoint.

   (c) Το Sidecar ξεκινάει και επεξεργάζεται τα αρχεία ρυθμίσεων του Μέλους με τις σωστές τιμές.

   (d) Το Sidecar ξεκινάει τη διεργασία της Cassandra.

Με αυτόν τον σχεδιασμό, μπορούμε να μεγαλώσουμε ένα Cassandra Cluster είτε αυξάνοντας τα Μέλη ενός Rack, είτε προσθέτοντας επιπλέον Rack σε ένα Datacenter.

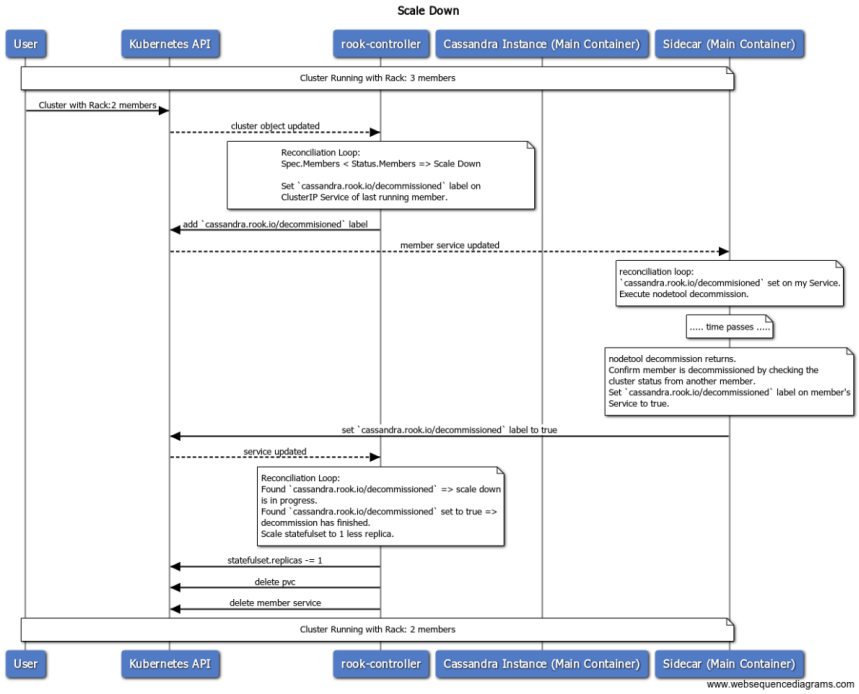
0.3.3.2 Σμίκρυνση ενός Cassandra Cluster

Όπως περιγράψαμε και στο προηγούμενο κεφάλαιο, η σμίκρυνση ενός Cassandra Cluster είναι μια διμερής διαδικασία:

1. Ο διαχειριστής θέλει να βγάλει ένα Μέλος. Του δίνει την εντολή να βγει (`nodetool decommission`).

2. Το Μέλος λέει στα άλλα μέλη ότι θέλει να φύγει. Γίνεται ανακατανομή των tokens και τα δεδομένα στέλνονται στα Μέλη που είναι τώρα υπεύθυνα για αυτά.

3. Το Μέλος βγαίνει από το Cassandra Cluster.



Σχήμα 10

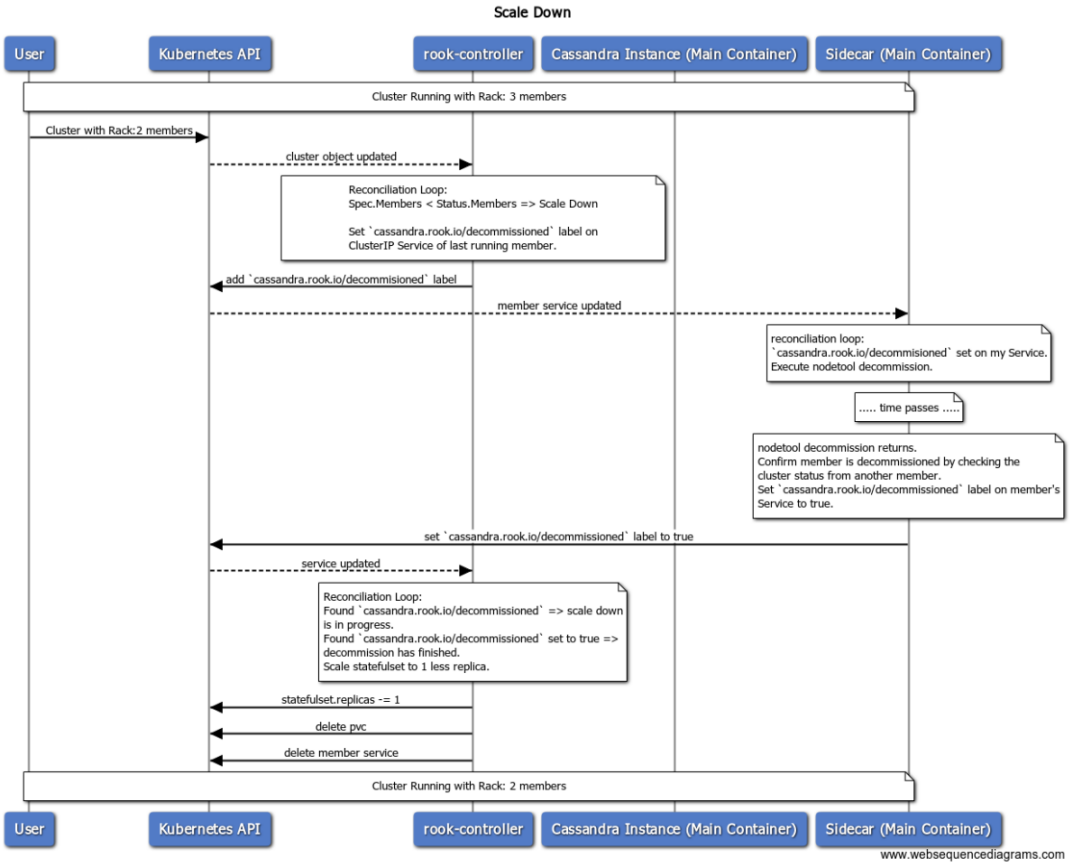
Πρέπει λοιπόν, με κάποιο τρόπο, να πούμε στο Μέλος ότι θέλουμε να φύγει. Πώς θα το κάνουμε όμως αυτό?

Μια λύση είναι η χρήση ενός HTTP API σε κάθε Μέλος, που θα μπορεί να πάρει εντολές από τον Controller. Όμως, μια τέτοια λύση παρουσιάζει διάφορα θέματα, καθώς η σμίκρυνση ενός Cassandra Cluster είναι μια χρονοβόρα διαδικασία και

αν συμβούν ενδιάμεσες διακοπές πρέπει να θυμόμαστε ότι το συγκεκριμένο Μέλος πρέπει να φύγει και να ολοκληρωθεί η διαδικασία.

Για τον λόγο αυτό, χρησιμοποιούμε έναν τρόπο επικοινωνίας πιο κοντά στα πρότυπα του Kubernetes. Για να πούμε στο Μέλος να φύγει, θα γράψουμε μια μόνιμη καταχώρηση που θα δηλώνει αυτή την επιθυμία μας. Αυτή η μόνιμη καταχώρηση θα είναι ένα label, το "cassandra.rook.io/decommissioned":"true", πάνω στο ClusterIP Service που αντιστοιχεί στο Μέλος.

Με βάση αυτά, η διαδικασία ενεργειών που ακολουθείται φαίνεται στο παρακάτω διάγραμμα:



Σχήμα 11: *Cluster Scale Down Sequence Diagram*

Επεξήγηση:

1. **Controller**: Παρατηρεί ότι ένα συγκεκριμένο Cassandra Rack πρέπει να μικρύνει (Rack[i].Spec.Members < RackStatus.Members).

2. **Controller**: Γράφει στο ClusterIP Service του Μέλους που πρέπει να φύγει το

label ”cassandra.rook.io/decommissioned”:”false”. Αυτό εκφράζει την επιθυμία μας να διώξουμε το συγκεκριμένο Μέλος.

3. **Sidecar**: Παρατηρεί το label στο ClusterIP Service του Μέλους και του λέει να φύγει από το Cluster τρέχοντας την εντολή nodetool decommission.

4. **Sidecar**: Αφού η εντολή επιστρέψει, επιβεβαιώνει ότι το Μέλος έχει φύγει και γράφει το label ”cassandra.rook.io/decommissioned”:”true” στο Service του Μέλους.

5. **Controller**: Παρατηρεί ότι το Μέλος έχει αποχωρήσει και μπορεί με ασφάλεια να μειώσει τον αριθμό Pods του StatefulSet, το οποίο με τη σειρά του το διαγράφει.

6. **Controller**: Διαγράφει επίσης το ClusterIP Service και τον δίσκο (Persistent Volume Claim) που αντιστοιχεί στο Μέλος.

0.4 Υλοποίηση

Σε αυτό το κεφάλαιο θα δώσουμε μία αναλυτική περιγραφή της υλοποίησής του Cassandra Operator, με εκτεταμένα παραδείγματα πηγαίου κώδικα.

0.4.1 Software Stack

Μία από τις πρώτες αποφάσεις που έπρεπε να πάρουμε ήταν τα εργαλεία που θα χρησιμοποιήσουμε για να γράψουμε τον Cassandra Operator, δηλαδή επιλογή γλώσσας προγραμματισμού, βιβλιοθηκών και frameworks.

- **Γλώσσα προγραμματισμού**: αποφασίσαμε να χρησιμοποιήσουμε την **Go** ως τη γλώσσα συγγραφής του Cassandra Operator. Αυτή η απόφαση πάρθηκε επειδή όλος ο Kubernetes είναι γραμμένος σε Go, οι περισσότεροι Operators είναι γραμμένοι σε Go και όλες οι καλές βιβλιοθήκες για να γράφεις Controllers είναι επίσης σε Go.

- **Libraries**: Θα χρησιμοποιήσουμε την **client-go** βιβλιοθήκη του Kubernetes, η οποία μας παρέχει έναν client για να μιλάμε στο Kubernetes API αλλά και πιο προχωρημένες δυνατότητες που θα αναφέρουμε αργότερα.

- **Frameworks**: Τη στιγμή συγγραφής του Cassandra Operator δεν υπήρχε κάποιο καλό framework για να γράψεις έναν Kubernetes Operator. Η κατάσταση αυτή έχει βελτιωθεί από τότε και πλέον υπάρχουν 2 πολύ καλά frameworks για να χρησιμοποιήσει κάποιος που γράφει Operators, το kubebuilder και το operator-sdk.

- **Collaborations**: Αποφασίσαμε την δουλειά που κάναμε στον Cassandra Operator, να τη συνεισφέρουμε στο project ανοιχτού λογισμικού Rook.io. Το Rook είναι ένα δημοφιλές λογισμικό για τη διαχείριση εφαρμογών αποθήκευσης δεδομένων στον Kubernetes, που περιλαμβάνει λύσεις για το Ceph, την CockroachDB και άλλα. Αποφασίσαμε να συνεργαστούμε με το Rook επειδή διέθετε:

  - Υγιή κοινότητα χρηστών και developers.
  - Testing Framework με ενσωμάτωση για Jenkins.
  - Code reviews από επαγγελματίες μηχανικούς λογισμικού με μεγάλη εμπειρία στο να τρέχουν συστήματα αποθήκευσης δεδομένων πάνω στον Kubernetes.
  - Κώδικα που μπορούμε να επαναχρησιμοποιήσουμε για τον δικό μας Operator.

0.4.2 Εσωτερική Αρχιτεκτονική Ενός Controller

Αφού αποφασίσαμε ποια γλώσσα, βιβλιοθήκη και framework θα χρησιμοποιήσουμε, ήρθε η ώρα να υλοποιήσουμε τον Controller μας. Αρχικά, θα παρουσιάσουμε το κυρίαρχο πρότυπο για την υλοποίηση Controllers, το οποίο είναι αυτό που θα ακολουθήσουμε.

Η γενική ιδέα των Controllers είναι ότι παρακολουθούν τα Αντικείμενα που τους ενδιαφέρουν, ειδοποιούνται όταν αυτά τα Αντικείμενα αλλάξουν και εκτελούν ενέργειες ώστε η πραγματική κατάσταση του συστήματος να ταυτιστεί με την επιθυμητή κατάσταση του χρήστη.

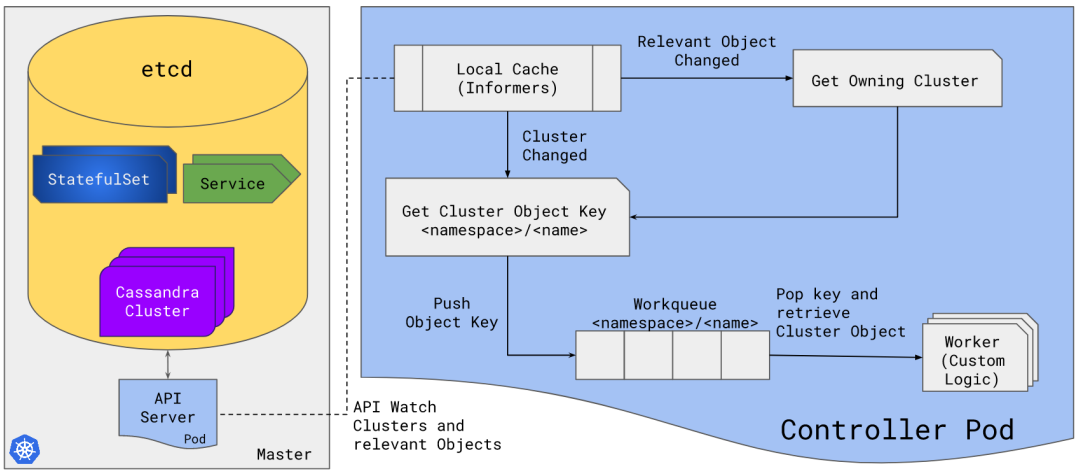
Στην πράξη, χρησιμοποιούνται μερικές ακόμα βελτιστοποιήσεις. Συγκεκριμένα, η βιβλιοθήκη client-go, χρησιμοποιώντας τη δομή Informer, κρατάει τοπικές caches των Αντικειμένων που παρακολουθεί τις οποίες κρατάει συγχρονισμένες με τα Αντικείμενα στον Kubernetes.

1. Χρησιμοποιούμε αυτές τις δομές για να παρακολουθούμε τα Αντικείμενα Cassandra Cluster καθώς και τα Αντικείμενα StatefulSet και Service που φτιάχνουμε.

2. Όταν αλλάζει ένα Cassandra Cluster, φτιάχνουμε ένα κλειδί από το αντικείμενο, της μορφής <namespace>/<name> και το βάζουμε σε μία ουρά (workqueue).

3. Όταν αλλάζει ένα StatefulSet, Service ή άλλο Αντικείμενο που παρακολουθούμε, βρίσκουμε το Cassandra Cluster στο οποίο ανήκει και βάζουμε το κλειδί αυτού του Cluster στην ουρά.

4. Από την άλλη μεριά της ουράς, υπάρχουν πολλές διεργασίες-νήματα (goroutines) οι οποίες αφαιρούν κλειδιά από την ουρά και εκτελούν τις ενέργειες που χρειάζονται. Όλη η λογική του Controller μας βρίσκεται σε αυτό το σημείο.



Σχήμα 12

0.4.3 Υλοποίηση του Σκελετού του Controller

Με βάση την εσωτερική αρχιτεκτονική ενός Controller, που περιγράψαμε παραπάνω, θα υλοποιήσουμε ένα σκελετό για τον Cassandra Cluster Controller.

0.4.3.1 Ορισμός του Cassandra Cluster Αντικειμένου

Πρώτα απ'όλα, θα γράψουμε σε κώδικα τη δομή που πρέπει να ακολουθεί ένα Cassandra Cluster Αντικείμενο. Τα αντικείμενα τύπου Cassandra Cluster θα αναπαρίστανται με αυτό το struct.

pkg/apis/cassandra.rook.io/v1alpha1/types.go

```
package v1alpha1

import (
    rook "github.com/rook/rook/pkg/apis/rook.io/v1alpha2"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

const (
    APIVersion = CustomResourceGroup + "/" + Version
)

//
↪ ******************************************************************************
// IMPORTANT FOR CODE GENERATION
// If the types in this file are updated, you will need to run
// `make codegen` to generate the new types under the client/clientset
↪ folder.
//
↪ ******************************************************************************

// Kubernetes API Conventions:
//
↪ https://github.com/kubernetes/community/blob/af5c40530f50c3b36c13438187b311102093e
// Applicable Here:
//  * Optional fields use a pointer to correctly handle empty values.
```

```
// +genclient
// +genclient:noStatus
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type Cluster struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata"`
    Spec              ClusterSpec   `json:"spec"`
    Status            ClusterStatus `json:"status"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type ClusterList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata"`
    Items           []Cluster `json:"items"`
}

// ClusterSpec is the desired state for a Cassandra Cluster.
type ClusterSpec struct {
    // Version of Cassandra to use.
    Version string `json:"version"`
    // Repository to pull the image from.
    Repository *string `json:"repository,omitempty"`
    // Mode selects an operating mode.
    Mode ClusterMode `json:"mode,omitempty"`
    // Datacenter that will make up this cluster.
    Datacenter DatacenterSpec `json:"datacenter"`
    // User-provided image for the sidecar that replaces default.
    SidecarImage *ImageSpec `json:"sidecarImage,omitempty"`
}
```

```
type ClusterMode string

const (
    ClusterModeCassandra ClusterMode = "cassandra"
    ClusterModeScylla    ClusterMode = "scylla"
)

// DatacenterSpec is the desired state for a Cassandra Datacenter.
type DatacenterSpec struct {
    // Name of the Cassandra Datacenter. Used in the
    ↪  cassandra-rackdc.properties file.
    Name string `json:"name"`
    // Racks of the specific Datacenter.
    Racks []RackSpec `json:"racks"`
}

// RackSpec is the desired state for a Cassandra Rack.
type RackSpec struct {
    // Name of the Cassandra Rack. Used in the
    ↪  cassandra-rackdc.properties file.
    Name string `json:"name"`
    // Members is the number of Cassandra instances in this rack.
    Members int32 `json:"members"`
    // User-provided ConfigMap applied to the specific statefulset.
    ConfigMapName *string `json:"configMapName,omitempty"`
    // Storage describes the underlying storage that Cassandra will
    ↪  consume.
    Storage rook.StorageScopeSpec `json:"storage"`
    // Placement describes restrictions for the nodes Cassandra is
    ↪  scheduled on.
    Placement *rook.Placement `json:"placement,omitempty"`
    // Resources the Cassandra Pods will use.
    Resources corev1.ResourceRequirements `json:"resources"`
```

```
}

// ImageSpec is the desired state for a container image.
type ImageSpec struct {
    // Version of the image.
    Version string `json:"version"`
    // Repository to pull the image from.
    Repository string `json:"repository"`
}

// ClusterStatus is the status of a Cassandra Cluster
type ClusterStatus struct {
    Racks map[string]*RackStatus `json:"racks,omitempty"`
}

// RackStatus is the status of a Cassandra Rack
type RackStatus struct {
    // Members is the current number of members requested in the
    ↪  specific Rack
    Members int32 `json:"members"`
    // ReadyMembers is the number of ready members in the specific Rack
    ReadyMembers int32 `json:"readyMembers"`
}
```

0.4.3.2 Δημιουργία μιας Εφαρμογής Γραμμής Εντολών

Ο Cassandra Controller ανήκει σε μια εφαρμογή της γραμμής εντολών (CLI application) του rook. Διαφορετικές υποεντολές ξεκινούν τους διάφορους controllers (πχ ”rook cassandra operator”, ”rook ceph operator”, ”rook ceph agent”). Εμείς θα ορίσουμε τον Controller μας ώστε να τρέχει με την εντολή ”rook cassandra operator”. Για τη δημιουργία του CLI χρησιμοποιείται η βιβλιοθήκη cobra.

cmd/rook/cassandra/operator.go

```
import (
    "github.com/rook/rook/cmd/rook/rook"
    rookinformers
    ↪ "github.com/rook/rook/pkg/client/informers/externalversions"
    kubeinformers "k8s.io/client-go/informers"
    "github.com/rook/rook/pkg/util/flags"
    "github.com/spf13/cobra"
    ...
)

const resyncPeriod = time.Second * 30

var operatorCmd = &cobra.Command{
    Use:   "operator",
    Short: "Runs the cassandra operator to deploy and manage cassandra
    ↪ in Kubernetes",
    Long: `Runs the cassandra operator to deploy and manage cassandra
    ↪ in kubernetes clusters.
https://github.com/rook/rook`,
}

func init() {
    flags.SetFlagsFromEnv(operatorCmd.Flags(), rook.RookEnvVarPrefix)

    operatorCmd.RunE = startOperator
}

func startOperator(cmd *cobra.Command, args []string) error {
...

    kubeClient, _, rookClient, err := rook.GetClientset()
    if err != nil {
```

```
        rook.TerminateFatal(fmt.Errorf("failed to get k8s clients.
        ↪  %+v\n", err))
    }

    kubeInformerFactory :=
    ↪  kubeinformers.NewSharedInformerFactory(kubeClient,
    ↪  resyncPeriod)
    rookInformerFactory :=
    ↪  rookinformers.NewSharedInformerFactory(rookClient,
    ↪  resyncPeriod)

    c := controller.New(
        rookImage,
        kubeClient,
        rookClient,
        rookInformerFactory.Cassandra().V1alpha1().Clusters(),
        kubeInformerFactory.Apps().V1().StatefulSets(),
        kubeInformerFactory.Core().V1().Services(),
        kubeInformerFactory.Core().V1().Pods(),
        kubeInformerFactory.Core().V1().ServiceAccounts(),
        kubeInformerFactory.Rbac().V1().Roles(),
        kubeInformerFactory.Rbac().V1().RoleBindings(),
    )

    // Create a channel to receive OS signals
    stopCh := server.SetupSignalHandler()

    // Start the informer factories
    go kubeInformerFactory.Start(stopCh)
    go rookInformerFactory.Start(stopCh)

    // Start the controller
    if err = c.Run(1, stopCh); err != nil {
```

```
        logger.Fatalf("Error running controller: %s", err.Error())
    }

    return nil
```

Όταν ξεκινάει λοιπόν ο Cluster Controller, δημιουργεί ένα νέο ClusterController struct το οποίο ρυθμίζει τους Informers ώστε να παρακολουθούν τα αντικείμενα που μας ενδιαφέρουν και να κρατάνε τοπικές caches για αυτά:

pkg/operator/cassandra/controller/controller.go

```
// ClusterController encapsulates all the tools the controller needs
// in order to talk to the Kubernetes API
type ClusterController struct {
    rookImage              string
    kubeClient             kubernetes.Interface
    rookClient             rookClientset.Interface
    clusterLister          listersv1alpha1.ClusterLister
    clusterListerSynced    cache.InformerSynced
    statefulSetLister      appslisters.StatefulSetLister
    statefulSetListerSynced cache.InformerSynced
    serviceLister          corelisters.ServiceLister
    serviceListerSynced    cache.InformerSynced
    podLister              corelisters.PodLister
    podListerSynced        cache.InformerSynced

    // queue is a rate limited work queue. This is used to queue work
    ↪  to be
    // processed instead of performing it as soon as a change happens.
    ↪  This
    // means we can ensure we only process a fixed amount of resources
    ↪  at a
    // time, and makes it easy to ensure we are never processing the
    ↪  same item
```

```
    // simultaneously in two different workers.
    queue workqueue.RateLimitingInterface
    // recorder is an event recorder for recording Event resources to
    ↪  the Kubernetes API
    recorder record.EventRecorder
}

// New returns a new ClusterController
func New(
    rookImage string,
    kubeClient kubernetes.Interface,
    rookClient rookClientset.Interface,
    clusterInformer informersv1alpha1.ClusterInformer,
    statefulSetInformer appsinformers.StatefulSetInformer,
    serviceInformer coreinformers.ServiceInformer,
    podInformer coreinformers.PodInformer,
    serviceAccountInformer coreinformers.ServiceAccountInformer,
    roleInformer rbacinformers.RoleInformer,
    roleBindingInformer rbacinformers.RoleBindingInformer,
) *ClusterController {

    // Add sample-controller types to the default Kubernetes Scheme so
    ↪  Events can be
    // logged for sample-controller types.
    rookScheme.AddToScheme(scheme.Scheme)
    // Create event broadcaster
    logger.Infof("creating event broadcaster...")
    eventBroadcaster := record.NewBroadcaster()
    eventBroadcaster.StartLogging(logger.Infof)
    eventBroadcaster.StartRecordingToSink(&typedcorev1.EventSinkImpl{Interface:
    ↪  kubeClient.CoreV1().Events("")})
    recorder := eventBroadcaster.NewRecorder(scheme.Scheme,
    ↪  corev1.EventSource{Component: controllerName})
```

```
cc := &ClusterController{
    rookImage:  rookImage,
    kubeClient: kubeClient,
    rookClient: rookClient,

    clusterLister:          clusterInformer.Lister(),
    clusterListerSynced:    clusterInformer.Informer().HasSynced,
    statefulSetLister:      statefulSetInformer.Lister(),
    statefulSetListerSynced:
    ↪  statefulSetInformer.Informer().HasSynced,
    podLister:              podInformer.Lister(),
    podListerSynced:        podInformer.Informer().HasSynced,
    serviceLister:          serviceInformer.Lister(),
    serviceListerSynced:    serviceInformer.Informer().HasSynced,

    queue:
    ↪  workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(
    ↪  clusterQueueName),
    recorder: recorder,
}

// Add event handling functions

clusterInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        newCluster := obj.(*cassandrav1alpha1.Cluster)
        cc.enqueueCluster(newCluster)
    },
    UpdateFunc: func(old, new interface{}) {
        newCluster := new.(*cassandrav1alpha1.Cluster)
        oldCluster := old.(*cassandrav1alpha1.Cluster)
```

```
            // If the Spec is the same as the one in our cache, there
            ↪  aren't
            // any changes we are interested in.
            if reflect.DeepEqual(newCluster.Spec, oldCluster.Spec) {
                return
            }
            cc.enqueueCluster(newCluster)
        },
        DeleteFunc: func(obj interface{}) {
            // TODO: handle deletion
        },
    })

    statefulSetInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: cc.handleObject,
        UpdateFunc: func(old, new interface{}) {
            newStatefulSet := new.(*appsv1.StatefulSet)
            oldStatefulSet := old.(*appsv1.StatefulSet)
            // If the StatefulSet is the same as the one in our cache,
            ↪  there
            // is no use adding it again.
            if newStatefulSet.ResourceVersion ==
            ↪  oldStatefulSet.ResourceVersion {
                return
            }
            // If ObservedGeneration != Generation, it means that the
            ↪  StatefulSet controller
            // has not yet processed the current StatefulSet object.
            // That means its Status is stale and we don't want to
            ↪  queue it.
            if newStatefulSet.Status.ObservedGeneration !=
            ↪  newStatefulSet.Generation {
                return
```

```
            }
            cc.handleObject(new)
        },
        DeleteFunc: cc.handleObject,
    })

    serviceInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            service := obj.(*corev1.Service)
            if service.Spec.ClusterIP == corev1.ClusterIPNone {
                return
            }
            cc.handleObject(obj)
        },
        UpdateFunc: func(old, new interface{}) {
            newService := new.(*corev1.Service)
            oldService := old.(*corev1.Service)
            if oldService.ResourceVersion == newService.ResourceVersion
            ↪  {
                return
            }
            cc.handleObject(new)
        },
        DeleteFunc: func(obj interface{}) {
            // TODO: investigate if further action needs to be taken
        },
    })

    return cc
}
```

Όπως είπαμε, αν αλλάξει ένα Cassandra Cluster Αντικείμενο βάζουμε το κλειδί του στην ουρά (enqueueCluster) ενώ αν αλλάξει ένα παρεμφερές Αντικείμενο που πα-

ρακολουθούμε, βρίσκουμε το Cluster Αντικείμενο στο οποίο ανήκει και βάζουμε αυτό στην ουρά (handleObject).

pkg/operator/cassandra/controller/controller.go

```
// enqueueCluster takes a Cluster resource and converts it into a
↪  namespace/name
// string which is then put onto the work queue. This method should not
↪  be
// passed resources of any type other than Cluster.
func (cc *ClusterController) enqueueCluster(obj
↪  *cassandrav1alpha1.Cluster) {
    var key string
    var err error
    if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
        runtime.HandleError(err)
        return
    }
    cc.queue.AddRateLimited(key)
}
```

pkg/operator/cassandra/controller/controller.go

```
// handleObject will take any resource implementing metav1.Object and
↪  attempt
// to find the Cluster resource that 'owns' it. It does this by looking
↪  at the
// objects metadata.ownerReferences field for an appropriate
↪  OwnerReference.
// It then enqueues that Cluster resource to be processed. If the
↪  object does not
// have an appropriate OwnerReference, it will simply be skipped.
func (cc *ClusterController) handleObject(obj interface{}) {
    var object metav1.Object
    var ok bool
    if object, ok = obj.(metav1.Object); !ok {
```

```
        tombstone, ok := obj.(cache.DeletedFinalStateUnknown)
        if !ok {
            runtime.HandleError(fmt.Errorf("error decoding object,
            ↪  invalid type"))
            return
        }
        object, ok = tombstone.Obj.(metav1.Object)
        if !ok {
            runtime.HandleError(fmt.Errorf("error decoding object
            ↪  tombstone, invalid type"))
            return
        }
        logger.Infof("Recovered deleted object '%s' from tombstone",
        ↪  object.GetName())
    }
    logger.Infof("Processing object: %s", object.GetName())
    if ownerRef := metav1.GetControllerOf(object); ownerRef != nil {
        // If the object is not a Cluster or doesn't belong to our
        ↪  APIVersion, skip it.
        if ownerRef.Kind != "Cluster" || ownerRef.APIVersion !=
        ↪  cassandrav1alpha1.APIVersion {
            return
        }

        cluster, err :=
        ↪  cc.clusterLister.Clusters(object.GetNamespace()).Get(ownerRef.Name)
        if err != nil {
            logger.Infof("ignoring orphaned object '%s' of cluster
            ↪  '%s'", object.GetSelfLink(), ownerRef.Name)
            return
        }

        cc.enqueueCluster(cluster)
```

```
        return
    }
}
```

0.4.3.3 Η Επανάληψη του Controller

Αφού δημιουργήθηκε το ClusterController struct και ο Cluster Controller παρακολουθεί τα αντικείμενα που τον ενδιαφέρουν, καλείται η συνάρτηση Run και αρχίζει η επανάληψη του Cluster Controller:

pkg/operator/cassandra/controller/controller.go

```
// Run starts the ClusterController process loop
func (cc *ClusterController) Run(threadiness int, stopCh <-chan
↪  struct{}) error {
    defer runtime.HandleCrash()
    defer cc.queue.ShutDown()

    //     Start the informer factories to begin populating the
    ↪  informer caches
    logger.Info("starting cassandra controller")

    // Wait for the caches to be synced before starting workers
    logger.Info("waiting for informers caches to sync...")
    if ok := cache.WaitForCacheSync(
        stopCh,
        cc.clusterListerSynced,
        cc.statefulSetListerSynced,
        cc.podListerSynced,
        cc.serviceListerSynced,
    ); !ok {
        return fmt.Errorf("failed to wait for caches to sync")
    }
```

```
    logger.Info("starting workers")
    for i := 0; i < threadiness; i++ {
        go wait.Until(cc.runWorker, time.Second, stopCh)
    }

    logger.Info("started workers")
    <-stopCh
    logger.Info("Shutting down cassandra controller workers")

    return nil
}

func (cc *ClusterController) runWorker() {
    for cc.processNextWorkItem() {
    }
}

func (cc *ClusterController) processNextWorkItem() bool {
    obj, shutdown := cc.queue.Get()

    if shutdown {
        return false
    }

    err := func(obj interface{}) error {
        defer cc.queue.Done(obj)
        key, ok := obj.(string)
        if !ok {
            cc.queue.Forget(obj)
            runtime.HandleError(fmt.Errorf("expected string in queue
            ↪  but got %#v", obj))
        }
        if err := cc.syncHandler(key); err != nil {
```

```
            cc.queue.AddRateLimited(key)
            return fmt.Errorf("error syncing '%s', requeueing: %s",
            ↪  key, err.Error())
        }
        cc.queue.Forget(obj)
        logger.Infof("Successfully synced '%s'", key)
        return nil
    }(obj)

    if err != nil {
        runtime.HandleError(err)
        return true
    }

    return true
}

// syncHandler compares the actual state with the desired, and attempts
↪  to
// converge the two. It then updates the Status block of the Cluster
// resource with the current status of the resource.
func (cc *ClusterController) syncHandler(key string) error {

    // Convert the namespace/name string into a distinct namespace and
    ↪  name.
    namespace, name, err := cache.SplitMetaNamespaceKey(key)
    if err != nil {
        runtime.HandleError(fmt.Errorf("invalid resource key: %s",
        ↪  key))
        return nil
    }

    // Get the Cluster resource with this namespace/name
```

```
    cluster, err := cc.clusterLister.Clusters(namespace).Get(name)
    if err != nil {
        // The Cluster resource may no longer exist, in which case we
        ↪  stop processing.
        if apierrors.IsNotFound(err) {
            runtime.HandleError(fmt.Errorf("cluster '%s' in work queue
            ↪  no longer exists", key))
            return nil
        }
        return fmt.Errorf("Unexpected error while getting cluster
        ↪  object: %s", err)
    }

    logger.Infof("handling cluster object: %+v", spew.Sdump(cluster))
    // Deepcopy here to ensure nobody messes with the cache.
    old, new := cluster, cluster.DeepCopy()
    // If sync was successful and Status has changed, update the
    ↪  Cluster.
    if err = cc.Sync(new); err == nil && !reflect.DeepEqual(old.Status,
    ↪  new.Status) {
        err = util.PatchClusterStatus(new, cc.rookClient)
    }

    return err
}
```

Με αυτό, ο σκελετός του Cluster Controller είναι έτοιμος.

0.4.4 Υλοποίηση της Λογικής του Cluster Controller

Μετά την ολοκλήρωση του σκελετού του Cluster Controller, αρχίζουμε να υλοποιούμε τις ακολουθίες ενεργειών για τα σενάρια που περιγράψαμε παραπάνω.

0.4.4.1 Δημιουργία και Μεγένθυση ενός Cassandra Cluster

Cluster Controller:

Ο κύκλος επανάληψης του Cluster Controller συνοψίζεται στην συνάρτηση Sync. Έτσι σε κάθε κύκλο επανάληψης, κάνουμε τις εξής ενέργειες:

- Δημιουργία, αν δεν υπάρχει, του Headless Service που θα συνδέονται οι clients.
- Δημιουργία, αν δεν υπάρχουν, των ClusterIP Services για κάθε Μέλος.
- Υπολογισμός του Cluster Status.
- Σύγκριση του Spec και Status και εκτέλεση των απαραίτητων ενεργειών. Αυτή η διαδικασία γίνεται μέσα στη συνάρτηση syncCluster.

pkg/operator/cassandra/controller/sync.go

```
// Sync attempts to sync the given Cassandra Cluster.
// NOTE: the Cluster Object is a DeepCopy. Modify at will.
func (cc *ClusterController) Sync(c *cassandrav1alpha1.Cluster) error {

    // Sync Headless Service for Cluster
    if err := cc.syncClusterHeadlessService(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageHeadlessServiceSyncFailed,
        )
        return err
    }

    // Sync Cluster Member Services
    if err := cc.syncMemberServices(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
```

```
            ErrSyncFailed,
            MessageMemberServicesSyncFailed,
        )
        return err
    }

    // Update Status
    if err := cc.updateStatus(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageUpdateStatusFailed,
        )
        return err
    }

    // Sync Cluster
    if err := cc.syncCluster(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageClusterSyncFailed,
        )
        return err
    }

    return nil
}
```

Ας εστιάσουμε τώρα την προσοχή μας στη συνάρτηση syncCluster, η οποία θα συγκρίνει το Spec με το Status και θα πάρει τις απαραίτητες ενέργειες.

pkg/operator/cassandra/controller/cluster.go

```
// SyncCluster checks the Status and performs reconciliation for
// the given Cassandra Cluster.
func (cc *ClusterController) syncCluster(c *cassandrav1alpha1.Cluster)
↪  error {

    // Check if any rack isn't created
    for _, rack := range c.Spec.Datacenter.Racks {
        // For each rack, check if a status entry exists
        if _, ok := c.Status.Racks[rack.Name]; !ok {
            logger.Infof("Attempting to create Rack %s", rack.Name)
            err := cc.createRack(rack, c)
            return err
        }
    }

    // Check that all racks are ready before taking any action
    for _, rack := range c.Spec.Datacenter.Racks {
        rackStatus := c.Status.Racks[rack.Name]
        if rackStatus.Members != rackStatus.ReadyMembers {
            logger.Infof("Rack %s is not ready, %+v", rack.Name,
            ↪  *rackStatus)
            return nil
        }
    }

    // Check if any rack needs to scale up
    for _, rack := range c.Spec.Datacenter.Racks {

        if rack.Members > c.Status.Racks[rack.Name].Members {
            logger.Infof("Attempting to scale rack %s", rack.Name)
            err := cc.scaleUpRack(rack, c)
            return err
```

```
        }
    }

    return nil
}
```

Για περισσότερες λεπτομέρειες και αναλυτικό κώδικα των επιμέρους συναρτήσεων, προτρέπουμε τον αναγνώστη να απευθυνθεί στο github repository του Rook, όπου έχουμε συνεισφέρει τον Cassandra Operator.

Sidecar:

Αφού υλοποιήσαμε τη λογική του Cluster Controller, ήρθε η ώρα να υλοποιήσουμε και τη διεργασία του Sidecar. Το Sidecar θα ξεκινάει επίσης ως υποεντολή του Rook binary, συγκεκριμένα με την "rook cassandra sidecar". Όταν ξεκινάει παίρνει τα δεδομένα που χρειάζεται από το API του Kubernetes, θα γράφει τις σωστές τιμές στα αρχεία ρυθμίσεων της Cassandra, και ξεκινάει τη διεργασία της Cassandra.

pkg/operator/cassandra/sidecar/sidecar.go

```
// onStartup is executed before the MemberController starts
// its sync loop.
func (m *MemberController) onStartup() error {

    // Setup HTTP checks
    m.logger.Info("Setting up HTTP Checks...")
    go func() {
        err := m.setupHTTPChecks()
        m.logger.Fatalf("Error with HTTP Server: %s", err.Error())
        panic("Something went wrong with the HTTP Checks")
    }()

    // Prepare config files for Cassandra
    m.logger.Infof("Generating cassandra config files...")
    if err := m.generateConfigFiles(); err != nil {
```

```
        return fmt.Errorf("error generating config files: %s",
        ↪  err.Error())
    }

    // Start the database daemon
    cmd := exec.Command(entrypointPath)
    cmd.Stderr = os.Stderr
    cmd.Stdout = os.Stdout
    cmd.Env = os.Environ()
    if err := cmd.Start(); err != nil {
        m.logger.Errorf("error starting database daemon: %s",
        ↪  err.Error())
        return err
    }

    return nil
}
```

0.4.4.2 Σμίκρυνση ενός Cassandra Cluster

Cluster Controller:

Για να υλοποιήσουμε την ακολουθία ενεργειών που χρειάζεται για τη σμίκρυνση ενός Cassandra Cluster, επεκτείνουμε τη συνάρτηση syncCluster.

pkg/operator/cassandra/controller/cluster.go

```
// SyncCluster checks the Status and performs reconciliation for
// the given Cassandra Cluster.
func (cc *ClusterController) syncCluster(c *cassandrav1alpha1.Cluster)
↪  error {

    // Check if any rack isn't created
    for _, rack := range c.Spec.Datacenter.Racks {
        // For each rack, check if a status entry exists
```

```
        if _, ok := c.Status.Racks[rack.Name]; !ok {
            logger.Infof("Attempting to create Rack %s", rack.Name)
            err := cc.createRack(rack, c)
            return err
        }
    }

    // Check if there is a scale-down in progress
    for _, rack := range c.Spec.Datacenter.Racks {
        if util.IsRackConditionTrue(c.Status.Racks[rack.Name],
        ↪  cassandrav1alpha1.RackConditionTypeMemberLeaving) {
            // Resume scale down
            err := cc.scaleDownRack(rack, c)
            return err
        }
    }

    // Check that all racks are ready before taking any action
    for _, rack := range c.Spec.Datacenter.Racks {
        rackStatus := c.Status.Racks[rack.Name]
        if rackStatus.Members != rackStatus.ReadyMembers {
            logger.Infof("Rack %s is not ready, %+v", rack.Name,
            ↪  *rackStatus)
            return nil
        }
    }

    // Check if any rack needs to scale down
    for _, rack := range c.Spec.Datacenter.Racks {
        if rack.Members < c.Status.Racks[rack.Name].Members {
            // scale down
            err := cc.scaleDownRack(rack, c)
            return err
```

```
        }
    }

    // Check if any rack needs to scale up
    for _, rack := range c.Spec.Datacenter.Racks {

        if rack.Members > c.Status.Racks[rack.Name].Members {
            logger.Infof("Attempting to scale rack %s", rack.Name)
            err := cc.scaleUpRack(rack, c)
            return err
        }
    }

    return nil
}
```

pkg/operator/cassandra/controller/cluster.go

```
// scaleDownRack handles scaling down for an existing Cassandra Rack.
// Calling this action implies all members of the Rack are Ready.
func (cc *ClusterController) scaleDownRack(r
↪  cassandrav1alpha1.RackSpec, c *cassandrav1alpha1.Cluster) error {

    logger.Infof("Scaling down rack %s", r.Name)

    // Get the current actual number of Members
    members := c.Status.Racks[r.Name].Members

    // Find the member to decommission
    memberName := fmt.Sprintf("%s-%d", util.StatefulSetNameForRack(r,
    ↪  c), members-1)
    logger.Infof("Member of interest: %s", memberName)
    memberService, err :=
    ↪  cc.serviceLister.Services(c.Namespace).Get(memberName)
```

```
    if err != nil {
        return fmt.Errorf("error trying to get Member Service %s: %s",
        ↪  memberName, err.Error())
    }

    // Check if there was a scale down in progress that has completed.
    if memberService.Labels[constants.DecommissionLabel] ==
    ↪  constants.LabelValueTrue {

        logger.Infof("Found decommissioned member: %s", memberName)

        // Get rack's statefulset
        stsName := util.StatefulSetNameForRack(r, c)
        sts, err :=
        ↪  cc.statefulSetLister.StatefulSets(c.Namespace).Get(stsName)
        if err != nil {
            return fmt.Errorf("error trying to get StatefulSet %s",
            ↪  stsName)
        }
        // Scale the statefulset
        err = util.ScaleStatefulSet(sts, -1, cc.kubeClient)
        if err != nil {
            return fmt.Errorf("error trying to scale down StatefulSet
            ↪  %s", stsName)
        }
        // Cleanup is done on each sync loop, no need to do anything
        ↪  else here

        cc.recorder.Event(
            c,
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackScaledDown, r.Name, members-1),
```

```
        )
        return nil
    }

    logger.Infof("Checking for scale down. Desired: %d. Actual: %d",
    ↪  r.Members, c.Status.Racks[r.Name].Members)
    // Then, check if there is a requested scale down.
    if r.Members < c.Status.Racks[r.Name].Members {

        logger.Infof("Scale down requested, member %s will
        ↪  decommission", memberName)
        // Record the intent to decommission the member
        old := memberService.DeepCopy()
        memberService.Labels[constants.DecommissionLabel] =
        ↪  constants.LabelValueFalse
        if err := util.PatchService(old, memberService, cc.kubeClient);
        ↪  err != nil {
            return fmt.Errorf("error patching member service %s: %s",
            ↪  memberName, err.Error())
        }

        cc.recorder.Event(
            c,
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackScaleDownInProgress, r.Name,
            ↪  members-1),
        )
    }

    return nil
}
```

Sidecar:

Στη διεργασία του Sidecar, χρειάζεται να παρακολουθούμε το ClusterIP Service του Μέλους και μόλις εντοπίσουμε το προσυμφωνημένο label, να αφαιρούμε το Μέλος από το Cassandra Cluster και να αλλάζουμε το label.

Για τον λόγο αυτό, χτίζουμε ένα σκελετό ανάλογο με αυτόν του Cluster Controller. Η συνάρτηση Sync είναι η εξής:

pkg/operator/cassandra/sidecar/sync.go

```
func (m *MemberController) Sync(memberService *v1.Service) error {

    // Check if member must decommission
    if decommission, ok :=
    ↪  memberService.Labels[constants.DecommissionLabel]; ok {
        // Check if member has already decommissioned
        if decommission == constants.LabelValueTrue {
            return nil
        }
        // Else, decommission member
        if err := m.nodetool.Decommission(); err != nil {
            m.logger.Errorf("Error during decommission: %s",
            ↪  err.Error())
        }
        // Confirm memberService has been decommissioned
        if opMode, err := m.nodetool.OperationMode(); err != nil ||
        ↪  opMode != nodetool.NodeOperationModeDecommissioned {
            return fmt.Errorf("error during decommission, operation
            ↪  mode: %s, error: %v", opMode, err)
        }
        // Update Label
        old := memberService.DeepCopy()
        memberService.Labels[constants.DecommissionLabel] =
        ↪  constants.LabelValueTrue
```

```
        if err := util.PatchService(old, memberService, m.kubeClient);
        ↪  err != nil {
            return fmt.Errorf("error patching MemberService, %s",
            ↪  err.Error())
        }

    }

    return nil
}
```

0.4.5 Πειραματική Αξιολόγηση και Αποτελέσματα

Σε αυτό το κεφάλαιο θα αξιολογήσουμε πειραματικά τον Cassandra Operator, δοκιμάζοντάς τον σε ένα Kubernetes Cluster.

0.4.5.1 Μεθολογία, Περιβάλλον και Εργαλεία

Το Kubernetes Cluster που θα χρησιμοποιήσουμε θα δημιουργηθεί από την πλατφόρμα Google Kubernetes Engine. Το script που χρησιμοποιούμε για τη δημιουργία του είναι το εξής:

gke-setup-script.sh

```
#!/bin/bash

set -e

GCP_USER=${1:-"yanniszarkadas@gmail.com"}
GCP_PROJECT=${2:-"gke-demo-226716"}

gcloud beta container clusters create "cluster" \
--project "$GCP_PROJECT" \
--region "europe-west1" \
```

```
--node-locations "europe-west1-b,europe-west1-c" \
--cluster-version "1.11.5-gke.5" \
--machine-type "n1-standard-4" \
--image-type "UBUNTU" \
--disk-type "pd-ssd" --disk-size "20" \
--local-ssd-count "1" \
--num-nodes "4" \
--no-enable-autoupgrade --no-enable-autorepair


# gcloud: Get credentials for new cluster
echo "Getting credentials for newly created cluster..."
gcloud container clusters get-credentials cluster --region=europe-west1

# Setup GKE RBAC
echo "Setting up GKE RBAC..."
kubectl create clusterrolebinding cluster-admin-binding --clusterrole
↪  cluster-admin --user "$GCP_USER"


echo "Checking if helm is present on the machine..."
if ! hash helm 2>/dev/null; then
  echo "You need to install helm. See:
  ↪  https://docs.helm.sh/using_helm/#installing-helm"
  exit 1
fi

# Setup Tiller
echo "Setting up Tiller..."
helm init
kubectl create serviceaccount --namespace kube-system tiller
kubectl create clusterrolebinding tiller-cluster-rule
↪  --clusterrole=cluster-admin --serviceaccount=kube-system:tiller
```

```
kubectl patch deploy --namespace kube-system tiller-deploy -p
↪  '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'

# Wait for Tiller to become ready
until [[ $(kubectl get deployment tiller-deploy -n kube-system -o
↪  'jsonpath={.status.readyReplicas}') -eq 1 ]];
do
    echo "Waiting for Tiller pod to become Ready..."
    sleep 5
done

# Install local volume provisioner
echo "Installing local volume provisioner..."
helm install --name local-provisioner provisioner
echo "Your disks are ready to use."
```

Το script αυτό δημιουργεί ένα Kubernetes Cluster με 4 κόμβους σε 2 Availability Zones (eu-west1-b, eu-west1-c) και χρησιμοποιεί τοπικούς δίσκους.

Θα αλληλεπιδρούμε με το Kubernetes Cluster μέσω της εντολής **kubectl** και με το Cassandra Cluster μέσω της εντολής **nodetool**.

Στη συνέχεια, εγκαθιστούμε τον Cassandra Operator μέσω της εντολής `kubectl apply -f operator.yaml` η οποία εγκαθιστά τον Cassandra Operator στο Kubernetes Cluster με τη δημιουργία των εξής Αντικειμένων:

cassandra-operator.yaml

```
# Namespace where Cassandra Operator will live
apiVersion: v1
kind: Namespace
metadata:
  name: rook-cassandra-system

---
```

```yaml
# Cassandra Cluster CRD
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: clusters.cassandra.rook.io
spec:
  group: cassandra.rook.io
  names:
    kind: Cluster
    listKind: ClusterList
    plural: clusters
    singular: cluster
  scope: Namespaced
  version: v1alpha1
  # openapi validation omitted for brevity

---

# ClusterRole for cassandra-operator.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: rook-cassandra-operator
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - list
      - watch
      - delete
```

```
  - apiGroups:
      - ""
    resources:
      - services
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - persistentvolumeclaims
    verbs:
      - get
      - create
      - delete
  - apiGroups:
      - ""
    resources:
      - nodes
      - persistentvolumes
    verbs:
      - get
  - apiGroups:
      - apps
    resources:
      - statefulsets
    verbs:
      - "*"
  - apiGroups:
      - policy
    resources:
      - poddisruptionbudgets
    verbs:
      - create
```

```
  - apiGroups:
      - cassandra.rook.io
    resources:
      - "*"
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - create
      - update
      - patch
---
# ServiceAccount for cassandra-operator. Serves as its authorization
↪  identity.
apiVersion: v1
kind: ServiceAccount
metadata:
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# Bind cassandra-operator ServiceAccount with ClusterRole.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rook-cassandra-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-cassandra-operator
subjects:
```

```
- kind: ServiceAccount
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# cassandra-operator StatefulSet.
 apiVersion: apps/v1
 kind: StatefulSet
 metadata:
   name: rook-cassandra-operator
   namespace: rook-cassandra-system
   labels:
     app: rook-cassandra-operator
 spec:
   replicas: 1
   serviceName: "non-existent-service"
   selector:
     matchLabels:
       app: rook-cassandra-operator
   template:
     metadata:
       labels:
         app: rook-cassandra-operator
     spec:
       serviceAccountName: rook-cassandra-operator
       containers:
       - name: rook-cassandra-operator
         image: yanniszark/rook-cassandra-operator:meetup-presentation
         imagePullPolicy: "Always"
         args: ["cassandra", "operator"]
         env:
         - name: POD_NAME
           valueFrom:
             fieldRef:
```

```
            fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

0.4.5.2 Αποτελέσματα

Δημιουργία και Μεγένθυση ενός Cassandra Cluster:

Αρχικά, υποβάλουμε στον Kubernetes το εξής Cassandra Cluster, με 3 Μέλη στο Availability Zone europe-west1-b:

cluster.yaml

```
# Cassandra Cluster
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: europe-west1
    racks:
      - name: europe-west1-b
        members: 3
        storage:
          volumeClaimTemplates:
                - metadata:
                    name: rook-cassandra-data
                  spec:
                    storageClassName: local-disks
```

```
                resources:
                  requests:
                    storage: 350Gi
        resources:
          requests:
            cpu: 2
            memory: 8Gi
          limits:
            cpu: 2
            memory: 8Gi
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: failure-domain.beta.kubernetes.io/zone
                    operator: In
                    values:
                      - europe-west1-b
```

Στη συνέχεια, ο Cassandra Operator ειδοποιείται ότι υπάρχει ένα νέο Cluster Object και αρχίζει να εκτελεί ενέργειες προκειμένου να το δημιουργήσει και να το μεγαλώσει.

Με το εργαλείο kubectl παρακολουθούμε τα Αντικείμενα που δημιουργεί. Έτσι βλέπουμε το StatefulSet που δημιουργεί για το Rack europe-west1-b:

```
yannis@dev-pc:~$ kubectl get statefulsets
NAME                                          DESIRED   CURRENT   AGE
rook-cassandra-europe-west1-europe-west1-b    3         3         1h
```

Τα Pods που δημιουργούνται από το StatefulSet:

```
yannis@dev-pc:~$ kubectl get pods
NAME                                            READY   STATUS    RESTARTS   AGE
```

```
rook-cassandra-europe-west1-europe-west1-b-0   1/1     Running   0          1h
rook-cassandra-europe-west1-europe-west1-b-1   1/1     Running   0          1h
rook-cassandra-europe-west1-europe-west1-b-2   1/1     Running   0          1h
```

Τα ClusterIP Services που δημιουργεί για το κάθε Μέλος:

```
yannis@dev-pc:~$ kubectl get service
NAME                                           TYPE        CLUSTER-IP      EXTERNAL-IP
rook-cassandra-client                          ClusterIP   None            <none>
rook-cassandra-europe-west1-europe-west1-b-0   ClusterIP   10.31.255.200   <none>
rook-cassandra-europe-west1-europe-west1-b-1   ClusterIP   10.31.241.133   <none>
rook-cassandra-europe-west1-europe-west1-b-2   ClusterIP   10.31.243.96    <none>
```

Το Status και τα Events που ανανεώνει ο Cassandra Operator:

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
  Racks:
    Europe - West 1 - B:
      Members:        3
      Ready Members:  3
Events:               <none>
```

Στη συνέχεια, προσθέτουμε 1 ακόμη Rack, στο Availability Zone europe-west1-c, με την εντολή `kubectl edit clusters.cassandra.rook.io rook-cassandra`.

cluster.yaml

```yaml
# Cassandra Cluster
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
```

```
  mode: cassandra
  datacenter:
    name: europe-west1
    racks:
      - name: europe-west1-b
        members: 3
        storage:
          volumeClaimTemplates:
                - metadata:
                    name: rook-cassandra-data
                  spec:
                    storageClassName: local-disks
                    resources:
                      requests:
                        storage: 350Gi
        resources:
          requests:
            cpu: 2
            memory: 8Gi
          limits:
            cpu: 2
            memory: 8Gi
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: failure-domain.beta.kubernetes.io/zone
                    operator: In
                    values:
                      - europe-west1-b
      - name: europe-west1-c
        members: 2
```

```
    storage:
      volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: local-disks
                resources:
                  requests:
                    storage: 350Gi
    resources:
      requests:
        cpu: 2
        memory: 8Gi
      limits:
        cpu: 2
        memory: 8Gi
    placement:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - europe-west1-c
```

Από το Status και τα Events του Cluster Αντικειμένου, παρακολουθούμε την εξέλιξη του αιτήματός μας:

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
  Racks:
    Europe - West 1 - B:
```

```
      Members:        3
      Ready Members:  3
    Europe - West 1 - C:
      Members:        2
      Ready Members:  2
Events:
  Type    Reason  Age    From                  Message
  ----    ------  ----   ----                  -------
  Normal  Synced  5m15s  cassandra-controller  Rack europe-west1-c created
  Normal  Synced  5m15s  cassandra-controller  Rack europe-west1-c scaled up to 1 members
  Normal  Synced  4m36s  cassandra-controller  Rack europe-west1-c scaled up to 2 members
```

Δημιουργήσαμε λοιπόν ένα Cassandra Cluster που εκτείνεται σε 2 Availability Zones, ξεπερνώντας τον περιορισμό που μας επέβαλε το απλό StatefulSet.

Σμίκρυνση ενός Cassandra Cluster

Αφού μεγαλώσαμε το Cassandra Cluster μας σε 5 Μέλη, θα αφαιρέσουμε 2 για να το μικρύνουμε. Συγκεκριμένα, θα αφαιρέσουμε 2 Μέλη από το Rack europe-west1-b. Θα χρησιμοποιήσουμε και πάλι την εντολή `kubectl edit clusters.cassandra.rook.io rook-cassandra`.

Αφού αλλάξουμε τον αριθμό Μελών, παρατηρούμε και πάλι ότι ο Operator ανανεώνει το Status του Cassandra Cluster Αντικειμένου και μας ενημερώνει για την κατάσταση αυτή τη στιγμή:

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
  Racks:
    Europe - West 1 - B:
      Members:        1
      Ready Members:  1
    Europe - West 1 - C:
      Members:        2
      Ready Members:  2
Events:
```

```
Type    Reason  Age                    From                  Message
----    ------  ----                   ----                  -------
Normal  Synced  16m                    cassandra-controller  Rack europe-west1-c crea
Normal  Synced  16m                    cassandra-controller  Rack europe-west1-c scal
Normal  Synced  15m                    cassandra-controller  Rack europe-west1-c scal
Normal  Synced  9m32s (x3 over 9m53s)  cassandra-controller  Rack europe-west1-b scal
Normal  Synced  8m52s                  cassandra-controller  Rack europe-west1-b scal
Normal  Synced  8m29s (x8 over 8m52s)  cassandra-controller  Rack europe-west1-b scal
Normal  Synced  7m51s                  cassandra-controller  Rack europe-west1-b scal
```

Αντίστοιχα, παρατηρούμε ότι το ClusterIP Service του Μέλους που αφαιρείται έχει το label ”cassandra.rook.io/decomissioned”:”false”. Μόλις στείλει τα δεδομένα του στα υπόλοιπα Μέλη, το Μέλος αλλάζει το label σε ”true” και ο Operator το αφαιρεί από το Cluster.

Εν τέλει, όπως βλέπουμε και από την εσωτερική κατάσταση της Cassandra με την εντολή `nodetool status`, τα Μέλη έχουν αφαιρεθεί.

Υλοποιήσαμε λοιπόν τη διαδικασία σμίκρυνσης ενός Cassandra Cluster σωστά και όπως θα γινόταν από έναν άνθρωπο διαχειριστή.

0.4.6 Επίλογος

0.4.6.1 Συμπεράσματα

Συνολικά, η σχεδίαση και η υλοποίησή μας κατάφερε να ανταποκριθεί στις προσδοκίες που θέσαμε. Ενώ δεν προλάβαμε να υλοποιήσουμε όλη τη λειτουργικότητα που θα θέλαμε, έχουμε ολοκληρώσει τη σχεδίαση των περισσοτέρων λειτουργειών και καταφέραμε η δουλειά μας να γίνει αποδεκτή σε ένα μεγάλο open source project, το Rook. Επίσης, η δουλειά μας τροποποιήθηκε και συνεισφέρθηκε στη Scylla, ως ο επίσημος Scylla Operator.

Μέσα από την επαφή μας με τον Kubernetes, μελετήσαμε σε βάθος τους περιορισμούς που υπάρχουν για τη διαχείριση συστημάτων αποθήκευσης δεδομένων και βρήκαμε τρόπους να τους υπερβούμε για την Cassandra, που όμως είναι χρήσιμοι και για άλλα συστήματα αποθήκευσης δεδομένων. Ο Kubernetes είναι μια πολύ

καλή πλατφόρμα διαχείρισης εργασιών, έχει τη δυνατότητα να τρέχει σε οποιοδήποτε περιβάλλον και πιστεύουμε πως η επιλογή του αποδείχθηκε σωστή.

0.4.6.2 Μελλοντικές Δυνατότητες

Στο άμεσο μέλλον, θα θέλαμε να υλοποιήσουμε επιπλέον λειτουργικότητα για τον Cassandra Operator ώστε να αποτελεί μια πιο ολοκληρωμένη λύση. Πιο συγκεκριμένα, θα θέλαμε να προσθέσουμε:

- Backups & Restores
- Web UI
- Αυτοματοποιημένα Repairs με τον Cassandra Reaper
- Multi-Region Clusters

Αυτές οι προτάσεις έχουν δρομολογηθεί και θα υλοποιηθούν στο πλαίσιο του Rook project.

Εκτός αυτού, είναι πολύ σημαντικό να γίνουν βελτιώσεις στον ίδιο τον Kubernetes για να υποστηρίζει καλύτερα συστήματα αποθήκευσης δεδομένων. Συγκεκριμένα, θα ήταν καλό να υπήρχε:

- Καλύτερη Υποστήριξη για Τοπικούς Δίσκους
- Υποστήριξη για Multi-Region Kubernetes Clusters

1 Introduction

In this chapter, we outline the scope of our work. We first provide a quick overview of the problem at hand. Next, we illustrate our proposed solution and mention some types of applications that can benefit from it. We move on to briefly describe some existing solutions and their shortcomings. Finally, we present the structure of the document.

1.1 Problem Statement

From the early years of computing to the exponential datasets of today, storage systems have always played a central role. The first systems ran on a single node, so operating them was manageable. However, the needs have since changed and now storage systems are mostly distributed, with multiple instances running across a number of nodes. To make matters worse, a company will usually run multiple of those distributed systems, ie for different environments (dev, staging, production), clouds (on-prem, AWS, GCP, Azure) and geographic locations (US, Asia, Europe). This complexity can no longer be handled manually and requires automation of the usual management processes.

Apache Cassandra is a distributed, scalable, and fault-tolerant NoSQL storage system that combines DynamoDB's architecture with BigTable's data model. Cassandra is a robust and effective storage system, used at scale in many companies like Apple, Netflix and Facebook.

Kubernetes is a cross-cloud, extensible open-source platform for managing container-

ized workloads and services, that facilitates both declarative configuration and automation.

In this work, we leverage the Kubernetes platform to create an robust management layer for Cassandra, which works across clouds and facilitates complex management operations that were previously done by hand. In Kubernetes, programs that manage Stateful Services running in the cluster are called Operators.

1.2 Motivation

In this section we highlight the value of our contribution by mentioning specific types of use cases that could benefit from it. In other words, we construct an indicative list of example use cases that could leverage a self-managed Cassandra cluster.

- **Write-Heavy Applications**: the internal mechanism of Apache Cassandra (LSM trees) favors write-heavy applications and provides huge write throughput. Those applications include user metadata, IoT, logging and archiving. Our work enables those applications to easily have a Cassandra cluster available on-demand.
- **Cassandra DBA**: nowadays companies need to run their workloads across multiple clouds (on-prem, AWS, GCP, Azure) and environments (dev, staging, production). Managing all of this complexity manually is very tiring and repetitive work that is cloud-provider specific. Using our work, much of the repetitive tasks a DBA has to do are automated leaving time for more essential work.
- **Automation for other storage providers**: using our work and the problems we faced as a guide, other storage providers (ie MongoDB, Redis, Kafka) can leverage Kubernetes to create a management solution for their software.

In a time that companies run on multiple clouds and need multiple instances of Cassandra for different environments, there is a need for a cross-cloud, easy-to-use, robust and open-source management layer, which will automate the deployment, scaling, disaster recovery and backups/restores for Apache Cassandra.

1.3 Existing Solutions

Our implementation is not the first attempt to automate management operations for Cassandra. In this section we briefly describe other such approaches and mention their similarities and differences from our own.

1.3.1 Netflix Priam

Priam is a process/tool that runs alongside Apache Cassandra to automate the following:

- Backup and recovery (Complete and incremental)
- Token management
- Seed discovery
- Configuration
- Support AWS environment

Priam uses SimpleDB, a proprietary, managed AWS database, to store its data and supports multi-datacenter clusters by using public IPs in AWS. Priam has been used in production at Netflix for many years, making it a very robust solution. However, because it's primarily used at Netflix, it has the following downsides:

- **Lack of Documentation**: The documentation for deploying and managing Priam is very lacking, almost non-existent. In addition, there are no architecture or other documents that would help a developer get started with the project.
- **Vendor Lock-In**: Priam can only run on AWS. The database it uses for storing state, SimpleDB, is only available as an AWS offering. In addition, Priam requires auto-scaling groups (ASGs) to function correctly, which is an AWS-specific API. All of these restrict the project and make it very difficult for it to run on multiple vendors.

- **Old Cassandra Standards**: Priam was developed when the Cassandra project was still at a very early stage. Because of this, it needs to manage the ring tokens on its own. This adds a lot of complexity and a big limitation: in order for data to remain balanced between nodes, scaling the cluster means doubling or halving its size, which is overkill most of the times. A PR has been opened to add vnode functionality to Priam, but hasn't been merged for over a year[1].

All of these reasons make Priam a less-than-ideal solution for running Cassandra in production (unless you are Netflix).

1.3.2 Datastax OpsCenter

Datastax was the main contributor of the Cassandra project, offering OpsCenter, its management solution, for free until a few years ago. After that, Datastax released its own proprietary version of Cassandra, DSE, and OpsCenter now only works with that.

DataStax OpsCenter is an easy-to-use visual management and monitoring solution for DataStax Enterprise (DSE), the always-on data layer for real-time applications. With OpsCenter, you can quickly provision, upgrade, monitor, backup/restore, and manage your DSE clusters with little to no expertise.

OpsCenter is certainly a very well-tested, production-ready solution for running Cassandra. However, it is proprietary software and not something that can be used and extended freely by everyone.

1.3.3 Jetstack Navigator

Navigator is a Kubernetes Operator for managing common stateful services on Kubernetes. It leverages Kubernetes primitives to build a management layer for databases and has support for Apache Cassandra. The model introduced by Navigator was a major inspiration for the model we decided to follow, albeit more complex. While Navigator has very well-written code by people that are contributors in the Kuber-

[1]Priam PR for adding vnodes: `https://github.com/Netflix/Priam/pull/623`

netes project, it hasn't been maintained since the July of 2018. In addition, it includes less functionality than our work.

1.3.4 Instaclustr cassandra-operator

cassandra-operator is also an effort to build a management layer for Casssandra on top of Kubernetes. It is maintained by Instaclustr, the main company behind Cassandra development at the moment. It is written in Java, so it misses on the more advanced functionality of go libraries targeted specifically at Kubernetes Operator. In addition, there are differences in the architectural decisions made compared to our solution. Still, this is a very promising project by a company with a high level of influence over the development of Cassandra.

1.3.5 Vanilla Kubernetes Approaches

Kubernetes provides very powerful primitives, so there have been tries to run Cassandra using only core Kubernetes objects, like StatefulSets. However, these efforts soon hit a wall when trying to accomodate more complex and database-specific functionality like deploying multi-zone clusters, orchestrating a correct scale down, backups and restores.

1.4 Thesis Structure

The rest of the document is organized as follows:

- **Chapter 2**: presentation of the theoretical background and concepts that our work is founded upon.
- **Chapter 3**: analysis of the architecture of our solution and design decisions from a higher-level perspective.
- **Chapter 4**: demonstration of the focal points of our implementation, reference to the problems we faced during the development process, as well as the proposed workarounds, optimizations and testing.

- **Chapter 5**: experimental evaluation of our solution.
- **Chapter 6**: concluding remarks, suggested future improvements and alternative approaches.

2 Background

In this chapter we provide the key theoretical elements for the understanding of our work. First, we explain several fundamental principles from the area of containers, distributed and storage systems.

2.1 OS-Level Virtualization & Containers

2.1.1 Definition

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances [2]. Those instances are generally called containers and there have been various implementations, from BSD and chroot jails, to LXC[1] and Docker[2], which made the concept popular.

2.1.2 Docker

2.1.2.1 Intro

Docker is the software that made commoditized containers and made them popular, by offering advanced capabilities with a familiar and easy-to-use user interface. Docker builds on two kernel features:

[1]LinuX Containers: `https://linuxcontainers.org/`
[2]Docker`https://www.docker.com/`

- **Namespaces**: Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. Examples of resources include pid, ipc and network.

- **CGroups**: Cgroups are a feature of the Linux kernel that limit the usage of certain resources for specified processes. For example, using cgroups, a user can limit a process' CPU and Memory allocation, network and I/O bandwidth.

Those two features work together to give an isolated, flexible environment for a process to run. Docker builds on that and adds the concept of images.

2.1.2.2 Images

A Docker Image is a file usually containing a program (eg Cassandra, MySQL) with all its dependencies. Using a Docker image, a user can launch a Container from it. Essentially, Docker took the concept of images from the world of Virtual Machines and applied it to containers.

A Docker Image is made of several read-only layers that are stacked on top of each other, using a union-capable file system[3]. This way, common layers are reused between different images and a Copy-On-Write strategy is applied.

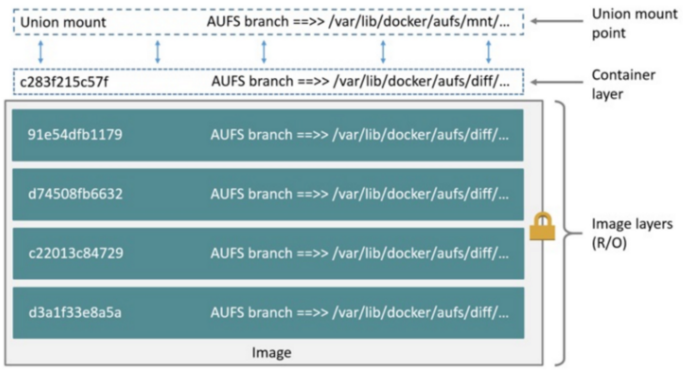


Figure 2.1: *Visualization of Docker Image Layers*

To create a Docker image, users can write a Dockerfile, which is a file with instructions of how to build an image written in a simple DSL[4]. For example:

[3] `https://en.wikipedia.org/wiki/Union_mount`
[4] `https://en.wikipedia.org/wiki/Domain-specific_language`

```
FROM ubuntu:latest

RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install Flask

ADD hello.py /home/hello.py

WORKDIR /home
```

This Dockerfile uses the ubuntu:latest image as a base image, installs Flask and adds a Flask application. Each one of those commands will propose a intermediary image layer and all of the layers will be stacked together to form the final image.

2.1.2.3 Docker Hub & Image Registries

After Docker introduced the concept of an image, it went even further by adding sharing and versioning capabilities by creating an image registry, Docker Hub. Much like how developers use Github to version code, they use Docker Hub to do the same for images. An image is versioned using tags, which are like git branches. For example, the ubuntu image can be found with the xenial tag or the bionic tag (ubuntu:xenial and ubuntu:bionic respectively).

2.1.2.4 Immutability & Reproducibility

By introducing the concept of immutable images containing a single application with its dependencies, Docker achieved true reproducability and cross-compatibility between different environments. If a computer has Docker, an image will run exactly the same way on it as any other computer with Docker. This enabled developers to have the same environment in their computer, to testing, staging and production. In addition, Docker Hub enables users to quickly find images for the software they want to use (eg Cassandra, Postgres) and developers to save, version and share their images.

2.1.2.5 Containers vs Virtual Machines

Many of the features listed so far also exist in virtual machines. However, virtual machines need to emulate a guest operating system, which is a big performance cost. On the contrary, containers reuse the host's kernel. This makes containers ideal, as they have instantaneous start time and can run on a range of machines, from low-resource development laptops as well as production servers.

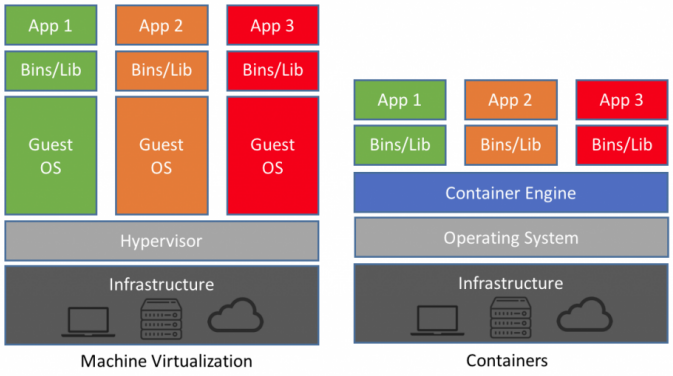


Figure 2.2: *Docker vs Virtual Machines Comparison*

Finally, the powerful and easy-to-learn DSL for Dockerfiles as well as the sharing and versioning features of Docker Hub, played a big role in making containers and Docker the standard way to deploy applications.

2.2 Kubernetes

2.2.1 Overview

With the rise of containers, apps run inside isolated, immutable, reproducible environments. Starting a new container is a very easy process that many developers go through on a regular basis. What happens though, when you have a lot of apps you want to run? You will need multiple physical nodes to run them all and someone needs to schedule containers on nodes and start them, execute health checks and restart unhealthy containers, scale apps by adding more containers, configure the network so containers can find each other easily and attach disks for storage requirements. Kubernetes is a platform for managing containerized workloads, a

system that solves all of the above problems and more.

2.2.2 Architecture

Kubernetes is designed with a Master-Follower architecture and is comprised of the Kubernetes Master and the Kubernetes Nodes. The Kubernetes Master, or Control Plane, contains the etcd database as well as the API server. The Kubernetes Nodes contain a number the kubelet for running containers and the kube-proxy for configuring Services. The Nodes run Pods, which are a wrapper around containers and the most basic building block of Kubernetes. The networking model of Kubernetes dictates that all Pods see themselves with the same IP as others see them and can contact each other and the Nodes without any form of NAT.

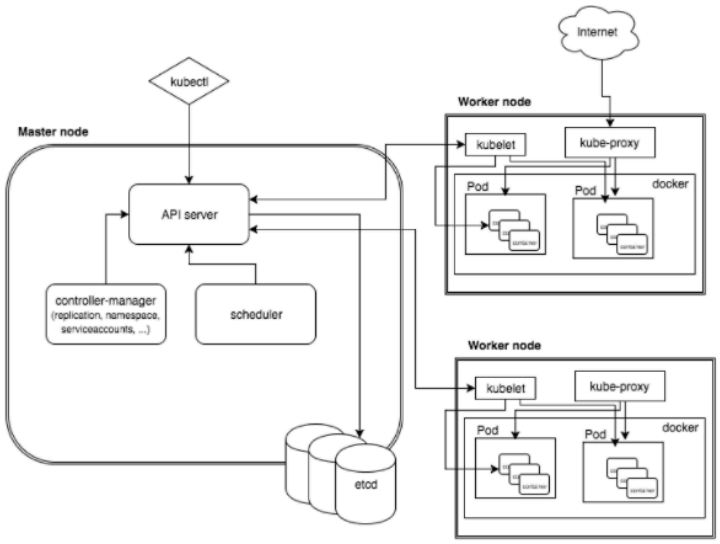


Figure 2.3: *Kubernetes Architecture Overview*

Let's take a closer look at what each component does:

Master:

1. **etcd**: The storage backend of Kubernetes. Every piece of data that must be persisted goes in etcd. etcd is a distributed key value store that provides a reliable way to store data across a cluster of machines. Even though Kubernetes uses a Master-Follower architecture which inherently represents a Single Point Of Failure (SPOF), it can achieve fault-tolerance by running multiple master nodes and using the distributed nature of etcd.

2. **API Server**: Kubernetes API server is the central management entity that receives all REST requests for modifications to Kubernetes Objects, serving as frontend to the cluster. Also, this is the only component that communicates with the etcd cluster.

3. **controller-manager**: Runs controller processes (explained below), which monitor Kubernetes Objects and regulate the cluster.

4. **scheduler**: Determines which Pod (group of containers) goes on each Node based on resource utilization.

Node:

1. **kubelet**: The main service of a Node, the kubelet watches the Kubernetes API Server to see if it needs to start/stop/resize any Pods. It also performs health-check on containers and restarts them if they're unhealthy. This component also reports to the master on the health of the host where it is running.

2. **kube-proxy**: Exposes the deployed container workloads to the end users/-clients.Each node runs a kube-proxy process which programs iptables rules to trap access to service IPs and redirect them to the correct backends.

2.2.3 Kubernetes Objects

Kubernetes Objects are persistent entities in the Kubernetes system, expressed in yaml[5] and stored in etcd. Kubernetes uses these entities to represent the state of the cluster. A Kubernetes object is a "record of intent" – once the user creates an object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's desired state.

2.2.3.1 Structure of an Object

Every object in Kubernetes has a set of common fields. Those fields are:

[5] `https://yaml.org/`

- **spec and status**: every Object in Kubernetes has a spec and a status object field. The spec describes the desired state of that object, while the status expresses the current state of an Object. For example, if the user deploys 3 instances of an web server but only 1 is created at the moment, the spec will show 3 while the status will show 1. This idea of spec and status in fundamental to Kubernetes and permeates all its design decisions, as we will discover later on.

- **apiVersion**: specifies the version of the Kubernetes API we're using to create the Object.

- **kind**: the kind of Object we want to create (eg. Pod, Deployment, Service).

- **metadata**: data that helps uniquely identify the Object. It includes the following subfields:

  - **uid**: A Kubernetes system-generated string to uniquely identify objects. Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID. It is intended to distinguish between historical occurrences of similar entities

  - **name**: the name of the Object. It is unique for the specific kind and namespace.

  - **namespace**: namespace the Object belongs in. Namespaces are essentially virtual clusters inside our Kubernetes cluster. They are used for multitenancy and they provide a scope for names.

  - **labels**: labels are arbitrary key-value pairs that are attached to objects. 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 (eg. "release":"stable", "environment":"dev", "tier":"backend"). Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion. Labels are not unique and multiple Objects may have the same labels. The Kubernetes API allows users to group objects together using a label selector. For example, a user can request all the Service Objects with label "tier":"backend".

- **annotations**: annotations are arbitrary key-value pairs that are attached to objects. The difference with labels is that selectors cannot be used on annotations, so Objects cannot be grouped together based on their annotations. In addition, annotations can hold much more data than labels, which have a 63 characters limitation. Example uses are build timestamps, image hashes, contact information of the person responsible, pointers to logging and analytic repositories.

Putting it all together, the basic structure of a Kubernetes object looks like the following:

```
apiVersion: v1
kind: Service
metadata:
  name: webserver
  namespace: default
  labels:
    app: nginx
  annotations:
    image-hash: e5sd4f6sd1f51sf8s4d1c5sf
  uid: s9or6nb5s2f64sd8f4th84
spec:
status:
```

After examining the common functionality across Kubernetes Objects, we introduce the core Kubernetes Objects and explain how the previous concepts materialize into code.

2.2.3.2 Pods

Pods are the smallest deployable objects in the Kubernetes API and the basic building blocks for many more complex Objects. A Pod is a wrapper around one or more

container(s), which form an application (eg Cassandra). A Pod also encapsulates storage, network identimintedty and runtime options for its containers.

Containers in the same Pod are automatically co-located and co-scheduled on the Node in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated. To better understand what a Pod is, let's see an example of a simple Pod:

```
apiVersion: v1
kind: Pod
metadata:
    name: web-app
spec:
  initContainers:
    - name: setup
      image: yanniszark/setup:1.0
  containers:
    - name: nodejs
      image: nodejs:8.0
      ports:
        containerPort: 7500
   - name: Database
     image: mongoDB:3.0
     ports:
       containerPort: 7501
     readinessProbe:
       httpGet:
         path: /healthz
         port: 8080
     readinessProbe:
       httpGet:
         path: /healthz
         port: 8081
```

In this example, we declare a Pod, named nodejs, with a nodejs web server container running alongside a MongoDB container. Nodejs is exposed on port 7500 and MongoDB on 7501 of the Pod's IP. Because containers on the same Pod are in the same network Linux namespace, they can communicate with each other using localhost, which creates a convenient and secure setup.

The MongoDB container also has two http probes. The kubelet will use those probes to find out if the container is alive and ready.

- **Liveness Probe**: A liveness probe determines if a container is alive, meaning the process is not dead or frozen.
- **Readiness Probe**: A readiness probe determines if a container is ready, meaning the process is running normally and is ready to serve requests.

Liveness and readiness probes can be HTTP probes, executables in the container or TCP sockets.

Before any of the two containers start, the initContainer setup will run first to do some configuration.

- **Init Containers**: InitContainers are containers that are run to completion before the main containers start. They are commonly used to do some setup before the main containers begin.

This very simple example illustrates the basic functions of a Pod.

After the Pod is created, it is given its own IP address, which other Pods in the cluster can use to contact it.

2.2.3.3 Services

Pods are ephemeral entities. They can die and not be resurrected, or their number may change. For example, we may have a nodejs server Pod that died and we recreated it. In this scenario, we want users and other Pods to still be able to find the new instance, without having to tell them explicitly.

In other words, someone has to keep track of the endpoints of our nodejs server, which can dynamically change over time. But the users of a Service shouldn't be

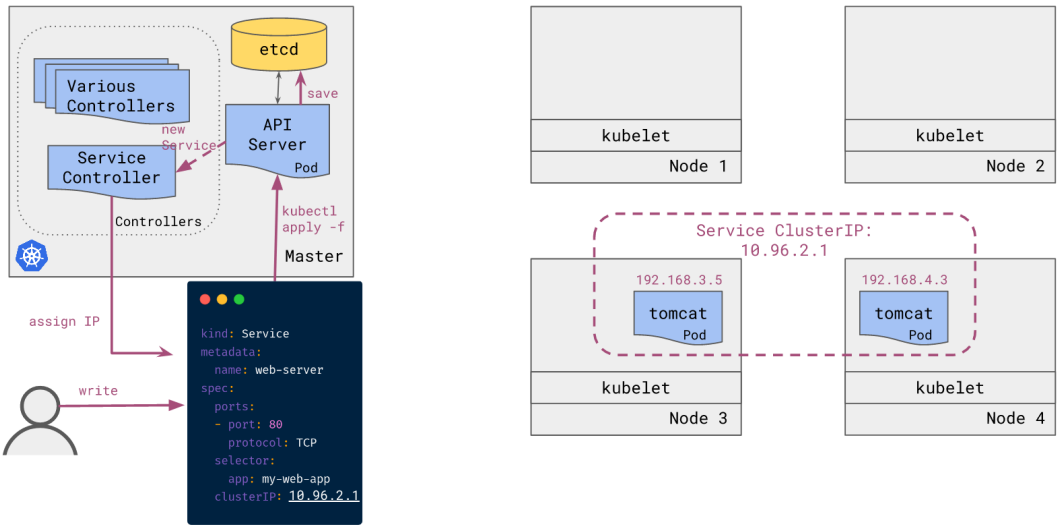


Figure 2.4: *Kubernetes Services*

the ones responsible for keeping track. The Service Object is Kubernetes' solution to enable this decoupling. For example:

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  clusterIP: 10.96.1.128
  selector:
    app: nodejs
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This Service Object tells Kubernetes that requests to the IP "10.96.1.128" must be routed to Pods with the label "app":"nodejs". The label constraint is specified by using a selector, which defines what constraints labels should satisfy. The selector is evaluated continuously and tracks the endpoints of the target Pods.

The Service also specifies a port. Whenever a request is made to the Service's clusterIP using the TCP protocol and port 80, it is redirected to port 9376 of the selected Pods. The Pods to which the Service will redirect our requests are also checked to be healthy and ready, as determined by the liveness and readiness probes.

2.2.3.4 Persistent Volumes, Claims and Storage Classes

Managing storage in Kubernetes is done with a couple of separate, complementary Objects. Kubernetes introduces an abstraction layer that allows for the decoupling of the implementation details of the storage layer (eg. local, NFS, GlusterFS, EBS, Ceph).

- **PersistentVolume**: A Persistent Volume (PV) is a piece of storage in the cluster that has been provisioned by an administrator. This API Object exposes fields common across storage providers as well as vendor-specific details about the implementation of the storage (eg GCEPersistentDisk, CephFS, Glusterfs). It has a lifecycle independent of the Pod(s) that uses it.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
```

```
  nfs:
    path: /tmp
    server: 172.17.0.2
```

- **PersistentVolumeClaim**: A Persistent Volume Claim (PVC) is a request for storage by a user. It is similar to a pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
```

Persistent Volume Claims allow users to consume abstract storage resources, without caring about the underlying implementation. However, many times users want these resources to have varying properties regarding QoS, like performance and availability. For partitioning the storage resources of a cluster arbitrarily according to specific properties, the StorageClass Object is used.

- **StorageClass**: A StorageClass provides a way for administrators to describe

the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
mountOptions:
  - debug
volumeBindingMode: Immediate
```

To better understand how PersistentVolumes (PVs), PersistentVolumeClaims(PVCs) and StorageClasses(SCs) interact, we examine the lifecycle of a Persistent Volume, after the user creates a PVC:

1. **Provisioning**: Provisioning of PVs can be either static or dynamic:
   - In *Static* provisioning, the PV Objects are created by an administrator and exist before the PVC is created by the user.
   - In *Dynamic* provisioning, the PVC is created and cannot be fulfilled, so a new PV is created to bind to that PVC. The StorageClass specified by the PVC must support dynamic provisioning for this to happen.
2. **Binding**: Once a PV with the requested capacity and access mode is provisioned, Kubernetes will bind it to the matching PVC. Once a PVC is bound, it is exclusive for the lifetime of the PVC. The PV to PVC binding is a one-to-one mapping.

3. **Using**: After the PV is bound to a PVC, it is mounted to Pods as a Volume. Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it.

4. **Reclaiming**: When the user has finished using the PV and no Pods are running that use the PVC, the user can delete the PVC Object to allow the reclamation of the PV resource. The PV is then reclaimed by Kubernetes, depending on its Reclaim Policy. The Reclaim Policy can be Retain or Delete, to keep the PV or delete the PV Object and the underlying asset (eg EBS volume, GooglePersistentDisk).

2.2.4 Controllers

We have explored the basic Kubernetes core primitives, which provide powerful abstractions and allow Kubernetes to work regardless of the underlying environment or cloud-provider. However, those primitives aren't very useful on their own. That's where Kubernetes Controllers come into play.

- **Controller**: A *Controller* is a piece of software that is responsible for a specific Kubernetes Object. It runs continuously and ensures that the *status* of an Object matches its **spec**, or that its *actual* state matches the *desired* state. The concept of Controllers is integral to the Kubernetes architecture. Kubernetes is actually a composition of loosely coupled programs, called Controllers, each responsible for an Object.

In other words, controllers are control loops that follow the general logic of:

1. Observe the current *Status* of the object

2. Analyze differences from desired *Spec* (eg. Pod "a" should be running but is not)

3. Act to reconcile the differences and achieve the desired Spec

Everything in Kubernetes is a Controller. Some examples include:

- **Scheduler**: The Kubernetes Scheduler is a Controller. It all watches the Pod Objects stored in etcd. If it finds any Pods that haven't been scheduled (the "nodeName" subfield of the spec is empty), then it will examine the Node Objects and try to find a proper fit for the Pod to run. Once it does, the scheduler creates a Binding Object, a *record of intent* to run the specific Pod on that specific Node.

- **kubelet**: The kubelet is a Controller. We could actually call it the Node Controller. Every kubelet in the cluster watches for Pod Objects in etcd that have nodeName equal to the node the kubelet is responsible for. If the kubelet finds such a Pod, it checks if it is running that Pod and if not it starts the Pod.

- **kube-proxy**: The kube-proxy is a Controller. We could call it the Service Controller. It watches for Service Objects in the etcd and creates ip-tables entries redirecting the traffic from the Service's Endpoints to the specified ClusterIP.

Besides using Controllers for the core functionality, Kubernetes uses them to provide extra functionality. Pods on their own are of little use, but when paired with Controllers, Kubernetes can provide extra functionality to cover a variety of workloads.

- **ReplicaSet**: ReplicaSet is the one of the most basic of Kubernetes Controllers. The ReplicaSet Controller has a very simple goal: to ensure a certain number of Pod replicas are running at any given time. Here is an example manifest for a ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
```

```yaml
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
```

The ReplicaSet controller works by watching ReplicaSet Objects. For every ReplicaSet it finds, it checks how many Pod replicas are running by using the specified Label Selector. If the number of Pod replicas is different than the number of desired replicas, the ReplicaSet creates or deletes Pods to ensure that the actual state matches the desired state.

- **Deployment**: Deployment is a higher level Controller that builds on ReplicaSets to provide advanced functionality, like Rolling Updates. Deployments are meant for running stateless applications that can scale without consideration for data safety.

```yaml
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

As we can see, the Spec of a Deployment is very similar to that of a ReplicaSet. The two controllers have similarities in their functionality, but Deployment is a more complete solution that provides the option to update the Spec of the Pod replicas or perform a rollback. Under the hood, Deployment leverages ReplicaSets to achieve that.

The Deployment Controller works in a similar fashion to the other controllers we have seen. It watches for Deployment Objects and for each Deployment found, it will take actions to ensure that the specified Spec matches the actual Status.

- **StatefulSet**: StatefulSets are the standard way of deploying stateful applications on Kubernetes. StatefulSet manages Pods based on a identical Pod Spec,

like Deployments. However, unlike Deployments, a StatefulSet maintains a sticky identity for each Pod. These Pods are created from the same Spec, but they are not interchangeable. The guarantees that StatefulSets provide are:

- Stable, unique network identifiers (A records, powered by a Headless Service).
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

To better understand how a StatefulSet works, let's take a look at a StatefulSet manifest:

```yaml
# Headless Service for network identity
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
# StatefulSet
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
```

```
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

We can spot many similarities with Deployments, like the Pod Spec. However, with StatefulSets, each Pod has its own persistent identity. StatefulSet Pods

are named with ordinal numbers appended to the name of the StatefulSet. For example, the StatefulSet ”web” with N replicas will create Pods ”web-0”,”web-1”,..., ”web_N-1”.

Regarding storage identity, we have the ability to add persistent storage to our Pods through the volumeClaimTemplates field. The StatefulSet controller will read the volumeClaimTemplates list and create a PersistentVolumeClaim per Pod for each list entry.

For network identity, we use the nginx Service, called a Headless Servuce because it doesn't have a clusterIP, which watches for Pods containing the ”app”:”nginx” label and creates DNS entries for them. This way, each Pod of the StatefulSet has a persistent network identity (eg for Pod ”web-0” in namespace ”default” the DNS entry will be ”web-0.default.svc”).

2.2.5 The Operator Pattern

We established that Kubernetes works with control loops called Controllers. Those Controllers watch Kubernetes Objects and observe the user-defined spec, calculate the real status and take actions to reconcile them. What if we could use the same pattern for an application management layer?

The Cassandra Cluster would be represented as a Kubernetes Object with a spec and a status. Then, we would create our own Controller, which would be responsible for Cassandra Cluster Objects. This is called the Operator Pattern, because the knowledge of a human operator about how to manage an application is put into a custom Controller which automates the whole process.

This still leaves us with a problem. How will we represent the Cassandra Cluster Object? Can we tell Kubernetes to store our custom Objects that we will be responsible for?

The mechanism that enables us to do this is called CustomResourceDefinition.

2.2.5.1 CustomResourceDefinition

CustomResourceDefinition is a Kubernetes Object that enables us to store our custom Objects in Kubernetes. This is the CustomResourceDefinition Object for the

Cassandra Cluster Objects:

Cassandra Cluster CRD

```yaml
# Cassandra Cluster CRD
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: clusters.cassandra.rook.io
spec:
  group: cassandra.rook.io
  names:
    kind: Cluster
    listKind: ClusterList
    plural: clusters
    singular: cluster
  scope: Namespaced
version: v1alpha1
```

2.3 Apache Cassandra

2.3.1 Overview

Apache Cassandra is a free and open-source, distributed, wide column store, NoSQL database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra was initially developed in 2008, at Facebook, by Avinash Lakshman and Prashant Malik, before being released and becoming an Apache Incubator project. It combines the data model of Google's BigTable and the replication model of DynamoDB.

2.3.2 Peer to Peer Design

Cassandra is a distributed database that uses a Peer-to-Peer, or masterless model. This means that every member in the cluster is equal to all other and there are no

special members. The Peer-to-Peer design of Cassandra makes it robust and removes any single point of failure from the system.

2.3.3 Data Model

In Cassandra, data is inserted and represented as *rows*. Rows are indexed by a *primary key* and organized into *tables*. Multiple tables form a *keyspace*. This data model is very close to what SQL provides, but in a Cassandra table, different rows may have some different rows. Because of this, Cassandra is often called a wide column store but that isn't exactly correct, because data isn't stored as columns on the disk.

Despite the data model resembling SQL very much, Cassandra doesn't support joins or sub-queries. Instead, it advocates for data denormalization, which means that data should be stored in the form it will be queried. This makes designing for Cassandra a unique problem, as you need have a good idea what your queries will be beforehand.

2.3.4 Topology of a Cassandra Cluster

A Cassandra cluster organizes members into the following topologies:

- **Datacenters**: A Cassandra cluster contains one or more Datacenters. A Datacenter is usually mapped to a region (eg. us-east-1).
- **Racks**: A Datacenter contains one or more Racks. A Rack is usually mapped to an availability zone (eg. us-east-1a).
- **Nodes(Members)**: A Node is the indivisible unit and building block of a Cassandra cluster. It represents a single computing unit in the Cassandra cluster. Because the name Node is the same as a Kubernetes Node, we will refer to Cassandra Nodes as Members.

2.3.5 Data Distribution Model

In a Cassandra cluster, data is distributed among members of a Datacenter. But how does a member know which pieces of data it should store? Cassandra solves this by using consistent hashing to achieve automatic sharding of its data.

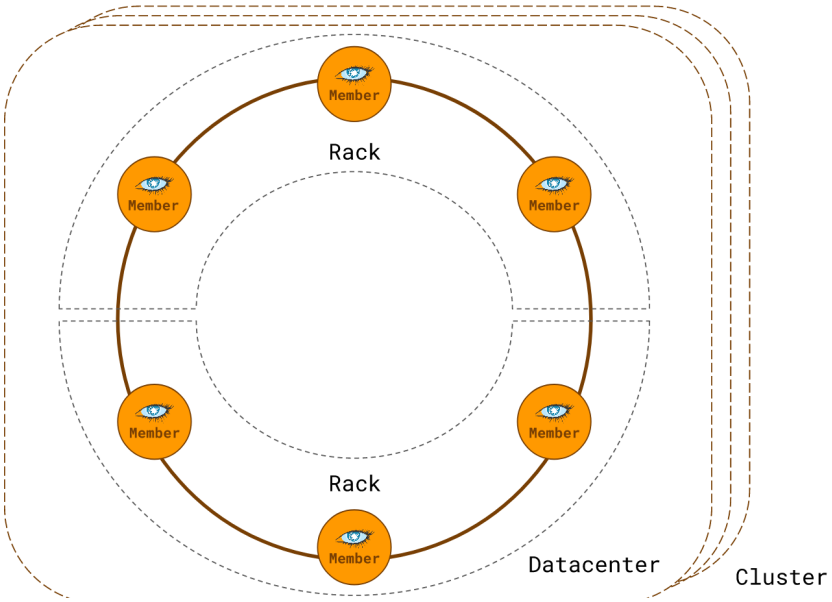


Figure 2.5: *Cassandra Cluster Topology*

Before a row is inserted, a subset of fields from the row, called the *partition key*, is given as input to a hash function that outputs a number in the range $0 - 2^{127-1}$. This range is called the *ring* because it is visualized as circle containing all the values. Each Cassandra member has *tokens*, which are points on this ring. The member owns the data from the hash value of its token to the start of another member's token.

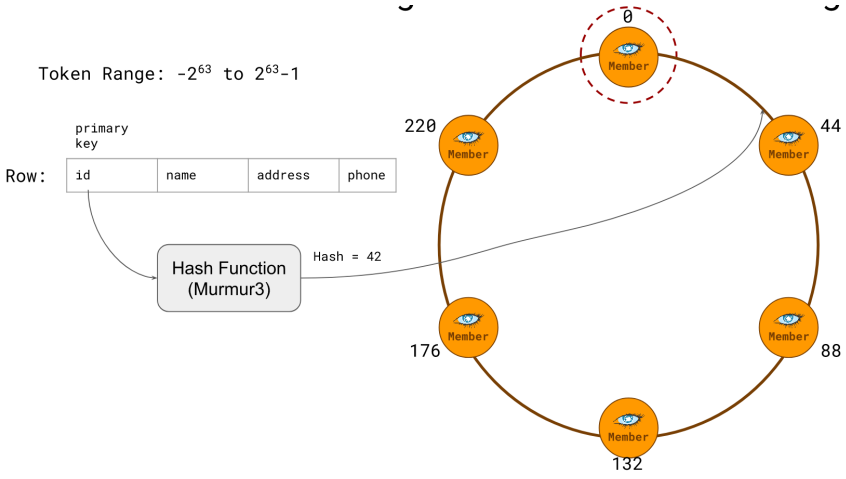


Figure 2.6: *Cassandra Data Partitioning with Tokens*

With that in mind, we can describe the procedure to insert a new row to a Cassandra cluster:

1. A subset of fields from the *primary key* of the table, called the *cluster key*, is

given as input to a hash function that outputs a number in the range $0-2^{127-1}$.

2. Given the tokens of each member, we walk the ring and find out which member is responsible for the particular hash.

3. The row is sent to the correct member to be persisted in the right partition.

The process followed to read a row follows a similar pattern:

1. A subset of fields from the *primary key* of the table, called the *cluster key*, is given as input to a hash function that outputs a number in the range $0-2^{127-1}$.

2. Given the tokens of each member, we walk the ring and find out which member is responsible for the particular hash.

3. The member searches the partition for the specified primary key and returns the requested row.

2.3.6 Replication, High Availability and Consistency

Cassandra replicates its data multiple times in order to provide fault tolerance. Each keyspace in Cassandra has a replication factor, which tells Cassandra how many replicas of the data it should keep.

Each Datacenter has a complete copy of the ring and as such contains all of the data. Inside a Datacenter, Cassandra replicates the data across members and racks.

Because of its peer-to-peer architecture, Cassandra can tolerate member failures and network partitions. In a split-brain scenario, each part of the cluster would operate independently of each other and reconcile when connectivity is returned.

Regarding consistency of the data, it is well known from the CAP Theorem that a distributed stateful system must choose between availability and consistency in the face of a network partition. Cassandra provides tunable consistency per operation, meaning that we can choose for each read/write operation how many replicas should answer to achieve the desired consistency level. This means that Cassandra can work as a CP or an AP database for different operations.

2.3.7 Internal Architecture

We have a high level view of how Cassandra distributes data across the cluster and how it handles a request to read/write data. However, we haven't said anything about the internals of how Cassandra writes and reads data, once a request arrives to a member.

Cassandra achieves high write throughput by making all writes of data be sequential writes. When a write request arrives, it is written to the following structures:

1. **Commit Log**: The *Commit Log* is an append-only log file that is written sequentially each time a write request arrives. It contains data in the order they were written.

2. **Memtable**: The *memtable* is an in-memory data structure. The data is written to the memtable in addition to the commit log. The memtable sorts the data written to it, so it always contains data in a sorted order and not the order they were written. The memtable also serves as a cache for read requests.

3. **SSTable**: After the Memtable has accumulated enough data, it is flushed to a structure called an SSTable, which exists on disk. The memtable is already sorted and is written sequentially on disk, which means it achieves high write throughput

   After the SSTable reaches a big enough size, a new SSTable is created. SSTables are immutable structures that isn't changed once it is written. The SSTables are organized into a structure called a Log Structured Merge Tree (LSM-Tree), which is a very popular data structure for turning random writes into sequential.

3 Design

3.1 Overview

In this chapter we thoroughly analyse the fundamental design decisions we made prior to the implementation of our solution. At first, we mention the expectations we want our final system to meet and present the architecture of the Cassandra Operator. We analyze the reasons behind the design and explore alternatives and why they were not selected.

Kubernetes Operators are a pattern for integrating Stateful applications in Kubernetes. The application is represented as a Custom Resource, a native Kubernetes Object, containing a Spec and Status field. The operator executes a control loop which get the desired Spec, observes the current Status and takes actions, much as a human operator would, to make sure that they match.

Our goal is to create a Kubernetes Operator for Apache Cassandra, which will automate the following common actions:

- Deployment of Cassandra clusters across availability zones
- Scaling Up
- Scaling Down
- Monitoring
- Recover from loss of a member

Our design also accommodates solutions for:

- Minor Version Upgrades
- Backups
- Restores
- Automated Repairs for Cassandra

3.2 Collaborations

3.2.1 Rook

When starting to design a solution for the Cassandra Operator, a decision had to be made: the operator would exist either as a standalone project or we would collaborate with an already existing project. After reviewing several projects, we decided to collaborate with the Rook project. [3]

Rook started as an Kubernetes Operator for running Ceph [4] on Kubernetes. It later expanded its scope to become a framework for creating Kubernetes Operators for storage providers. Collaborating with rook offered many advantages:

- Design for the Cassandra Operator will be peer reviewed by industry experts.
- A healthy community and userbase that will embrace, use and test the Cassandra Operator.
- End to end testing framework with Jenkins integration, which will help us easily add tests.
- Common reusable functionality to be leveraged by all storage providers.

The Rook community welcomed the effort from the beginning and has constantly provided help and guidance. We are especially grateful for the continuous help of Jared Watts, Bassam Tabbara, Travis Nielsen and Alexander Trost.

This chapter is heavily based on the design documents published publicly at the Rook project [1].

3.2.2 ScyllaDB

Scylla [5] is a close-to-the-hardware rewrite of Cassandra in C++. It features a shared nothing architecture that enables true linear scaling and major hardware optimizations that achieve ultra-low latencies and extreme throughput. It is a drop-in replacement for Cassandra and uses the same interfaces, so an operator for Cassandra should work for Scylla with little changes.

With this thought in mind, we made an attempt to approach the Scylla community and see if there is an interest in our effort. The Scylla community responded with enthusiasm and helped us with testing and technical questions. We want to especially thank Moreno Garcia and Juliana Oliveira for their continuous support.

3.3 Design Goals

While designing our solution for the Cassandra Operator, we always had certain goals in mind. The Cassandra Operator should:

- Be level-based in the Kubernetes sense, immune from edge-case race-condition scenarios.
- Provide a UX consistent with the Kuberneter API.
- Leverage the existing Kubernetes API Objects to offload as much work as possible. We don't want to reinvent the wheel.
- Provide an all-in-one production-ready solution to run Cassandra on Kubernetes.
- Allow for easy manual intervention when needed.

[1] Cassandra Operator Design Documentshttps://github.com/rook/rook/tree/df73f142bfa390e8b9cc8b24eb3a8d8058425dda/design/cassandra

3.4 Mapping of Concepts

As stated in the Background section about Cassandra (2.3), Cassandra abstracts resources in the following order:

Cluster -> Datacenter -> Rack -> Member (Node)

We map those abstractions to Kubernetes in the following way:

Cassandra	Kubernetes
Member	Pod
Rack	StatefulSet
Datacenter	Pool of StatefulSets
Cluster	Cluster CRD

We make the decision to map each Rack to a StatefulSet because that way, we can guarantee the placement of the Pods on the desired availability zone, with the use of the NodeAffinity feature of Kubernetes.

3.5 High Level Overview

Our design makes use of the operator pattern: a Custom Resource Definition is used to expose a Cassandra Cluster Kubernetes Object to the user and a Controller is deployed to make sure the actual state, for each Cluster, matches the desired state.

However, there are some operational actions in Cassandra that are hard or impossible to do remotely. Some of these actions are:

- **Initial Setup**: On each member's startup, we need to edit config files with the correct options in order to allow the member to correctly join the cluster and tune performance settings.
- **Backups/Restores**: For backups, we need to issue commands to the JMX administrative interface [6] of the member and then upload the backup to a remote location. For restores, we need to pull data from a remote location and place it in the correct paths.

To support these scenarios that require interaction with the local process and filesystem, we have two options:

1. **SSH-Tunneling**: The first option is to connect to an instance remotely via SSH. This option isn't preferable for the following reasons:
   (a) Difficult to handle programmatically. Instead of using native language functions, we have to use direct commands and parse output from stdout and stderr, making it difficult to correctly handle errors and to write simple, comprehensible code.
   (b) Fragile and easy to disrupt, especially for longer procedures like backup which may take a long time to complete.
2. **Sidecar**: Instead of trying to issue commands via SSH remotely and dealing with the aforementioned drawback, we propose adding a lightweight process, a sidecar, that will run alongside Cassandra and help with editing configuration files, taking backups, performing restores and execute health checks by talking to the JMX API that each member exposes. This has the following advantages over SSH tunneling:
   (a) Direct access to each member's filesystem, so it can edit configuration files easily.
   (b) Low latency, no-overhead, stable access to the Cassandra JMX API and the local filesystem.
   (c) Scales naturally along with the Cluster.
   (d) Able to handle complex failure scenarios.

After those observations, we decided to enrich the Operator pattern with a sidecar running alongside each Cassandra process. The following diagram is a visualization of the Cassandra Operator's design:

3.6 Controller-Sidecar Communication

A question that immediately arises from our decision to include a sidecar process alongside each Cassandra member, is how each sidecar will communicate with the

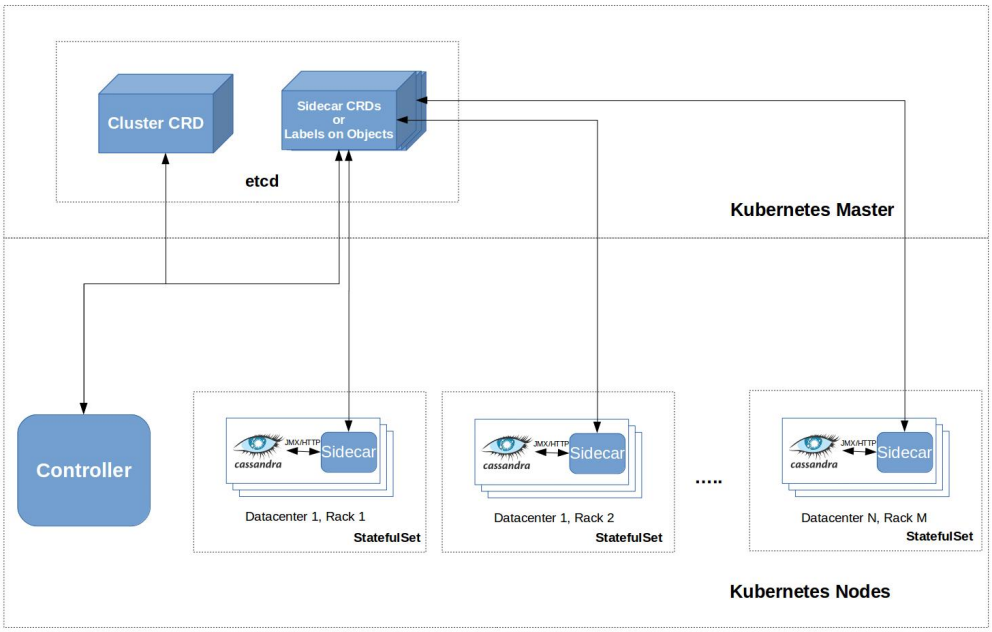


Figure 3.1: *Operator Design Overview*

Controller so it knows when to perform backups, restores, decommission of the member and other actions.

A solution that immediately comes to mind is to set up a REST HTTP API on the sidecar, which the Controller will use. This is the approach used by Netflix Priam.

However, this includes some extra complications. Remember that, according to our goals we are designing a level-based system. Many operations in Cassandra, like backups and decommissions, may take a long time to complete. Backup, for example, might take hours to complete. That means our operator must have an open TCP connection for all this time. If it gets interrupted, which we do expect to happen eventually, this connection will be lost and we won't have any record that it ever happened.

This doesn't seem like the Kubernetes way of doing things. The kubelet does a similar job of helping containers run in Nodes, so let's observe how the kubelet communicates with the Scheduler to know which Pods it has to run:

- **Question**: Does the scheduler ping the kubelet each time it schedules a Pod and then wait for an answer?

- **Answer**: No, it writes the nodeName field on the PodSpec. In other words, it writes a *record of intent*. No matter how many times the kubelet or the scheduler crashes it doesn't matter. The record of intent is there permanently.

Based on this observation, we will design a method of communication that is in-line with the Kubernetes philosophy:

When the Controller wants to invoke a functionality in a sidecar, it should write a *record of intent* in the Kubernetes API Objects (etcd). The sidecar will be watching the Kubernetes API and respond accordingly.

We examine two approaches to represent the record of intent:

1. **Labels**: When the controller wants to communicate with a sidecar, it will write a predefined label in the ClusterIP Service Object of the specific instance. For example, to communicate that we want an instance to de- commission, we could write the label 'cassandra.rook.io/decommission'. The sidecar will see this and decommission the Cassandra instance. When it is done, it will change the label value to a predefined value. Then the controller will know to delete that instance.

   Advantages

   (a) Reuses the Kubernetes built-in mechanisms.

   (b) Labels are query-able.

   (c) Simple solution.

   Disadvantages

   (a) Doesn't support nested fields.

   (b) Labels don't benefit from the strongly typed nature of Go.

   (c) Method of communication must be documented very clearly, as it isn't obvious how it works.

2. **Member CRD**: Each member will have a corresponding Member Custom Resource, which presents the Cassandra member as a Kubernetes Object, with a Spec and Status. Each sidecar will watch its Member Object and work to make

sure that requirements specified in the Spec are met. The Controller will write the Spec of each member to communicate the desired state, while the sidecar will update the Status and try to reconcile the two.

Advantages

(a) More expressive and natural.

(b) Supports nested fields.

(c) Only our operator touches it, which means less chance of a user accidentally changing an Object and affecting the Operator, compare to the Labels solution which will write Labels on a Service Object.

Disadvantages

(a) Over-complicated solution to have for a few fields.

(b) Induces an extra burden on etcd because we store more Objects.

(c) More complicated to implement in code.

After considering the advantages and disadvantages of each approach, we decided to start implementing the Cassandra Operator with the Labels approach, avoiding the extra complexity of the Member CRD approach. If in the future it becomes clear that the Member CRD approach is superior, the operator will be modified to follow that approach.

3.7 Member Identity

Mapping Racks to StatefulSets allows us to leverage much of the existing functionality of Kubernetes for running stateful workloads. This includes a stable storage and network identity, meaning they persist across Pod restart or deletion. The persistent network identity provided by StatefulSets is in the form of a DNS entry. For example, the ”member-1.default.svc” DNS entry will always point to the Pod ”member-1”.

In Cassandra, a Member’s identity is a UUID, called Host-ID, that is generated when the Member joins the Cluster. However, to replace a lost Member in a Cassandra Cluster, you have to specify the Member’s IP. The problem is that IPs in Kubernetes

are ephemeral and Cassandra doesn't accept a hostname. In addition, getting the Host-ID of a Member when it starts is prone to race-conditions. A Member can register with the Cluster and fail before we get a chance to know its Host-ID.

In other words, we need each Member to have an identity that we can specify beforehand, so in every case we can know which Member to replace. Doing bookkeeping with Host-IDs is possible but complicated and prone to the race-condition we specified above. While that race-condition isn't expected to be common, we believe it will happen eventually for big clusters.

Instead, we explore the possibility of having static IP addresses for Members. This would solve the problem of the Member identity, since each Member's identity would be its IP which we know beforehand. In addition, many Cassandra administrative commands work with IPs, it would be very convenient. With this in mind, we explore the following solution:

Service Per Member: Normally, a Service is used to expose a static IP (ClusterIP) that others can use to reach a set of Pods (eg Pods running web servers for a site). So Services already have what we need, a static IP, they are just meant to be used with multiple Pods.

What if we leveraged the static IP of a Service and created a ClusterIP Service per Pod? That would provide each Member of the Cluster with a static IP and thus a static identity that we know of beforehand. Each Service will select its corresponding Pod using the label "statefulset.kubernetes.io/pod-name" that is set by the StatefulSet Controller.

Before taking the decision, we contemplate the possible drawbacks of such an approach:

1. **Exhausting Service IPs**: Most clusters have a /12 IP block to utilize for giving out ClusterIPs, so IP space exhaustion isn't something that is expected to happen as a result of this approach.

2. **Performance**: A question that immediately comes to mind is how much overhead do the extra Services incur. To answer this, we look at how the Service functionality is implemented. According to the Kubernetes documentation [7] the default implementation is using IP-Tables. This type of implementa-

tion starts to suffer in the 1000s of Services. However, this high of a number is not very common and another implementation using IPVS doesn't suffer from this scalability issue, so we can conclude that there isn't any meaningful performance overhead.

While Services were not meant to be used that way, after careful investigation we decided that they are a good fit for our use case without any serious drawbacks.

3.8 Example Cluster CRD

After detailing the design for the Cassandra CRD, we can now write up an example Cassandra Cluster Custom Resource, the Object that our users will interact with:

```yaml
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  repository: my-private-repo.io/cassandra
  mode: cassandra
  datacenter:
    name: us-east-1
    racks:
      - name: us-east-1a
        members: 3
        storage:
          volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: my-storage-class
```

```
          resources:
            requests:
              storage: 200Gi
    resources:
      requests:
        cpu: 8
        memory: 32Gi
      limits:
        cpu: 8
        memory: 32Gi
    placement:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/region
                operator: In
                values:
                  - us-east-1
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - us-east-1a
```

The Object structure follows the pattern of Kubernetes as well as the guidelines set by Rook. We briefly explain the various settings for a Cluster CRD:

Cluster Settings:

- **version**: The version of Cassandra to use. It is used as the image tag to pull.
- **repository**: Optional field. Specifies a custom image repo. If left unset, the official docker hub repo is used.

- **mode**: Optional field. Specifies if this is a Cassandra or Scylla cluster. If left unset, it defaults to cassandra. Values: scylla, cassandra

In the Cassandra model, each cluster contains datacenters and each datacenter contains racks. At the beginning, the operator will only support single datacenter setups.

Datacenter Settings:

- **name**: Name of the datacenter. Usually, a datacenter corresponds to a region.
- **racks**: List of racks for the specific datacenter.

Rack Settings:

- **name**: Name of the rack. Usually, a rack corresponds to an availability zone.
- **members**: Number of Cassandra members for the specific rack. (In Cassandra documentation, they are called nodes. We don't call them nodes to avoid confusion as a Cassandra Node corresponds to a Kubernetes Pod, not a Kubernetes Node).
- **storage**: Defines the volumes to use for each Cassandra member. Currently, only 1 volume is supported.
- **resources**: Defines the CPU and RAM resources for the Cassandra Pods.
- **placement**: Defines the placement of Cassandra Pods. Has the following subfields:
  - **nodeAffinity**: Pin the Members of the Rack to specific Nodes [2]
  - **podAffinity**: Define rules to affect the scheduling of Members in the Rack with respect to other Pods (eg run as many Members as possible on the same Node).[3]

[2] `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity`

[3] `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity`

- **podAntiAffinity**: Define rules to avoid the scheduling of Members in the Rack with respect to other Pods (eg. don't run more than 1 Member on each Node) [4]
- **tolerations**: Define rules to allow scheduling on Nodes with taints. This feature can be used along with taints to mark specific Nodes to only be used for Cassandra.[5].

3.9 Workflow of Individual Actions

After finalizing our general design principles and decision, we begin planning the workflows and sequencing of events for individual actions, like Cluster creation, scale up, scale down and monitoring.

3.9.1 Cluster Creation & Scale Up

First of all, we describe the sequence for creating and scaling up a Cassandra cluster. The procedure of Cluster creation can be seen as starting from a Rack of 0 members and then scaling up one-by-one the Rack to the number of Members requested.

With that thought in mind, we plan out the sequencing of events for the creation of a Cassandra Cluster:

Explanation:

1. **User** creates a Cluster Custom Resource, requesting a Cassandra Cluster:

```
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
```

[4] https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity

[5] https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/

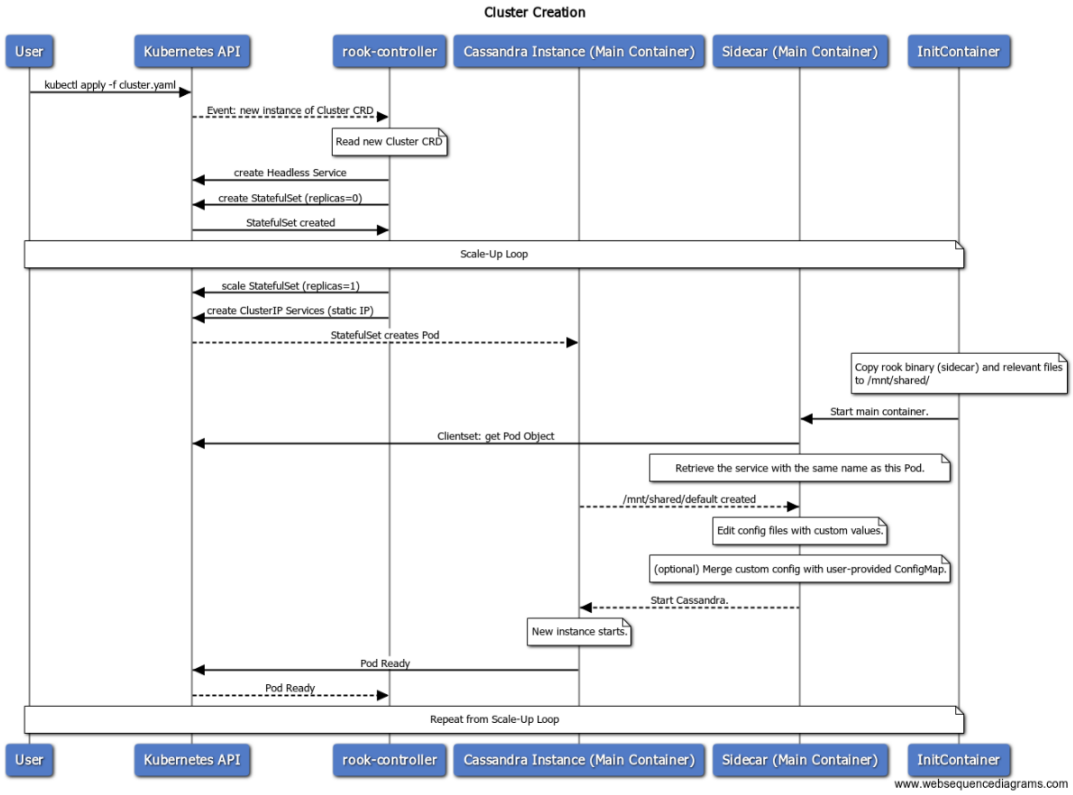


Figure 3.2: *Cluster Creation and Scale-Up Sequence Diagram*

```
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: us-east-1
    racks:
      - name: us-east-1a
        members: 3
        storage:
          volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: my-storage-class
                resources:
                  requests:
```

```
            storage: 200Gi
      resources:
        requests:
          cpu: 8
          memory: 32Gi
        limits:
          cpu: 8
          memory: 32Gi
      placement:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: failure-domain.beta.kubernetes.io/zone
                  operator: In
                  values:
                    - us-east-1a
```

2. **Controller** is informed of a new Cluster CRD instance. For each (dc,rack) it creates:

   (a) i. If a StatefulSet for the Rack doesn't exist, create it with 0 replicas.

   ii. If a StatefulSet for the Rack exists, check if the number of replicas equals the number of desired Members. If it needs more Members, increment the number of replicas in the StatefulSet Spec.

   (b) ClusterIP Services that serve as static IPs for Pods, labeled accordingly if they are seeds. Each Service is named after the Pod that uses it.

   The Controller also creates a Headless Service for the Cluster that clients will use to connect to Ready Members of the database.

3. **Member** (StatefulSet Pod) starts and runs an init container:

   (a) Init container starts and copies our custom sidecar binary and other necessary files (like plugins) to the shared volume /mnt/shared/, then exits.

(b) The Cassandra container starts with our custom binary (sidecar) as its entrypoint.

(c) Sidecar starts and edits config files with custom values applying to Kubernetes.

(d) Sidecar starts the Cassandra process and its own control loop that watches its ClusterIP Service.

With this design, we can scale up either by adding a new Rack or by adding Members to an existing Rack.

We decided to run the sidecar in the same container as the Cassandra process, by injecting the sidecar binary via an init container. This approach provides the following advantages:

1. Sidecar has direct access to the filesystem of the Cassandra Member, in order to edit config files in-place.

2. Separate containers means users need to define cpu and ram requests for the sidecar. This fragments the resources of a Node and provides a bad UX (ie if a Node has 16 cores, you want to give 16 to Cassandra, not 15.9 to Cassandra and 0.1 to the Sidecar). It also doesn't integrate well with the CPU Manager feature [6] for cpu affinity, that users may want to use for increased performance.

3.9.2 Scale Down

Scaling down a Cassandra Cluster is a two part procedure:

1. Issue a decommission command on the instance that will shut down. Once the instance receives this command, it will stream its data to the other instances and then shut down permanently. This process can be very lengthy, especially for large datasets.

2. Once the instance has decommissioned, the StatefulSet can be safely scaled down. The PVC of the deleted Pod will remain (default behaviour of State-

[6] Kubernetes CPU Manager for core pinning `https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/`

fulSet), so we need to clean it up or it may cause problems in the future (if the StatefulSet scales up again, the new Pod will get bound to the old PVC).

We remind that, in Cassandra operator, each Pod has a corresponding ClusterIP Service, that serves as a static IP and thus its identity. We also use labels on those objects to communicate intent in our operator. For example, the ”cassandra.rook.io/seed” label communicates that the instance is a seed. For database management and operations, Cassandra uses a Java RPC Interface (JMX). Since Go can't talk to JMX, we use an HTTP<->JMX bridge, Jolokia. This way, we can operate Cassandra though HTTP calls.

With that in mind, the algorithm we use for scaling down a Cassandra Rack is:

1. **Operator**: Detect requested scale down (Rack[i].Spec.Members < RackStatus.Members).
2. **Operator**: Add label cassandra.rook.io/decommissioned to the ClusterIP Service of the last pod of the StatefulSet. This serves as the *record of intent* to decommission that instance.
3. **Sidecar**: Detect the cassandra.rook.io/decommissioned label on the ClusterIP Service Object.
4. **Sidecar**: Run nodetool decommission on the Member.
5. **Sidecar**: Confirm that decommission has completed on the Member.
6. **Sidecar**: Update label to ”cassandra.rook.io/decommissioned”:”true”.
7. **Operator**: Detect label change and scale down the StatefulSet.
8. **Operator**: Delete PVC of the now-deleted Pod.

The following sequence diagram visualizes the aforementioned process:

3.9.3 Monitoring

Cassandra provides monitoring metrics through the JMX RPC interface. However, we want to use a modern monitoring solution, Prometheus, which is the standard way of monitoring applications on Kubernetes.

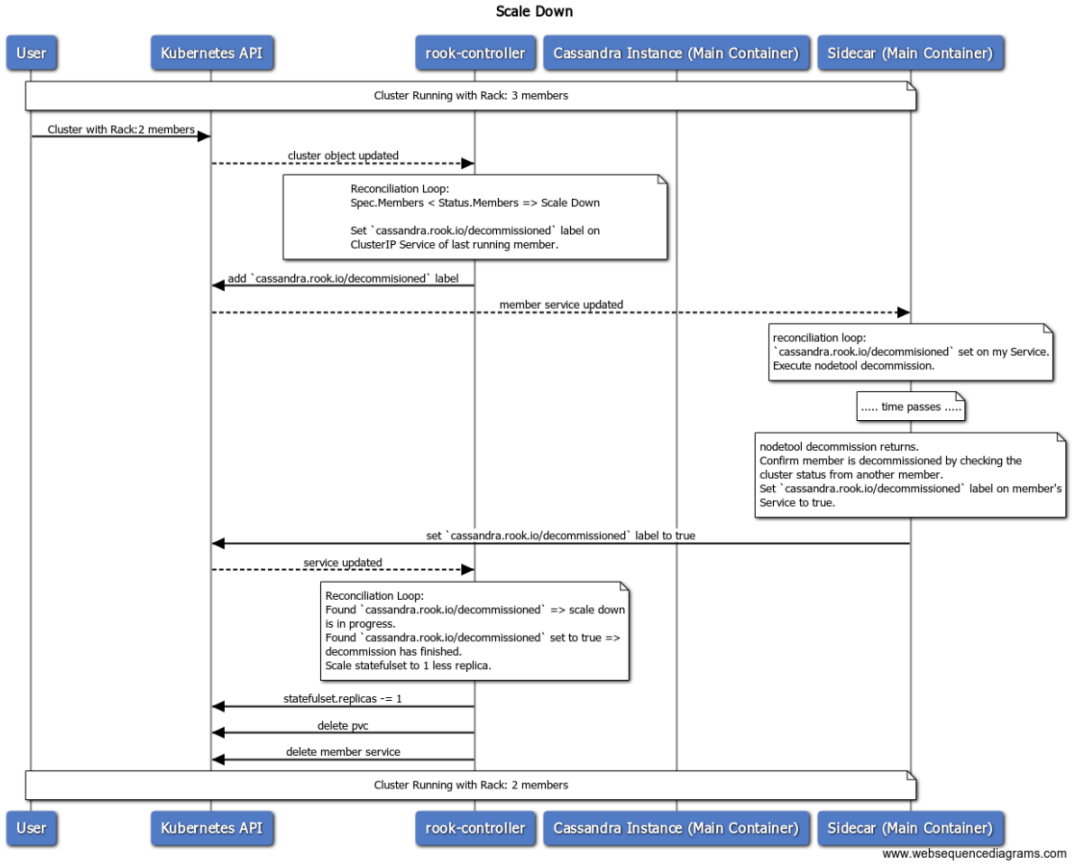


Figure 3.3: *Cluster Scale Down Sequence Diagram, as seen in subsection 3.9.2*

To expose Cassandra metrics in a Prometheus friendly way, we investigate open source solutions. We consider the following solutions:

1. **cassandra-exporter by Instaclustr**: Provides superior performance compared to other solutions. It is an early project, but is backed by the main company behind Apache Cassandra development. Can run as a javaagent or standalone (with decreased performance).

2. **cassandra-exporter by criteo**: A fork of jmx-exporter, focused on Cassandra metrics with advanced features like caching of expensive metrics. It is used in production by criteo and comes with its own Grafana dashboard. It is meant to be run as a standalone process, which leads to a decreased performance.

3. **jmx-exporter**: The oldest and most mature software. It has a broader scope and isn't limited only to Cassandra. Because of this, it may be harder to integrate. Can be run as a javaagent or standalone.

After careful consideration, we decided to go with cassandra-exporter, to take advantage of its superior performance and the fact that it is backed by a company with extensive knowledge of Apache Cassandra, that also uses it in its own Kubernetes Operator.

3.9.4 Anti-Entropy Repairs

Cassandra is designed to remain available if one of it's nodes is down or unreachable. However, when a node is down or unreachable, it needs to eventually discover the writes it missed. These inconsistencies are fixed with the repair process. Repair synchronizes the data between nodes by comparing their respective datasets for their common token ranges, and streaming the differences for any out of sync sections between the nodes. It compares the data with merkle trees, which are a hierarchy of hashes. [8]

Orchestrating repairs in a Cassandra Cluster is not a simple task. One cannot simply repair all data at once, because the increased load can bring the Cluster down. Fortunately, there is a good open-source solution that we can aim to integrate with our Kubernetes Operator, which can take care of repairs.

Cassandra Reaper: [9] is a centralized, stateful tool for running Apache Cassandra repairs against single or multi-site clusters. It was originally developed by *Spotify* [7] and is now maintained by *The Last Pickle* [8] , a company doing consulting for Apache Cassandra.

3.9.5 Local Disks

For the most part of its history, Kubernetes has focused on using Network Attached Storage solutions that provide replication and fault-tolerance of the Persistent Volumes. This fault-tolerance however, comes with a reduction in performance.

On the other hand, we have the solution of using fast, local disks with much better performance and cost. These local disks are ephemeral and not replicated.

[7]Spotify Company `https://www.spotify.com/`
[8]The Last Pickle Company `http://thelastpickle.com/`

Cassandra handles replication in the application level. Because of this, it is unnecessary to use slower and more expensive Network Attached Storage. Instead, we will try to use Local Storage with Kubernetes.

Local Storage can fail in the three following ways:

- **Disk Malfunctions**: using the disk gives block errors, performance is decreased but the Pods can still run.
- **Disk Fails**: Mount point of the volume / block device disappears. The Pod cannot start.
- **Node with Local Disk Fails**: The Pod cannot start.

We will tackle the third case, which is the most common case in the cloud, where VM migrations, upgrades and other things often cause data loss in local disks.

Let's see how a Node Failure would play out. Assume a Kubernetes environment with a Cassandra Cluster of 3 Members using local disks, each Member on a different Node. Node of member-1 fails, bringing the local disk down with it.

Administrator:

- Knows that the Node is failed and deletes the corresponding Node Object from Kubernetes.

Cassandra Operator:

- Detects that a Member's PersistentVolume has NodeAffinity for a Node that does not exist.
- Deletes the Member's PVC and recreates it, in order to give that Member a new empty disk.
- The member-1 Pod starts again.

The Cassandra Sidecar:

- Contacts another Member and finds out that member-1 is bootstrapped.

- Starts Cassandra with the JVM option -Dcassandra.replace_address_first_boot=<dead_-node_ip>.
- Cassandra detects that bootstrap data is missing, assumes the failed member-1's tokens and streams its data back from other Members.

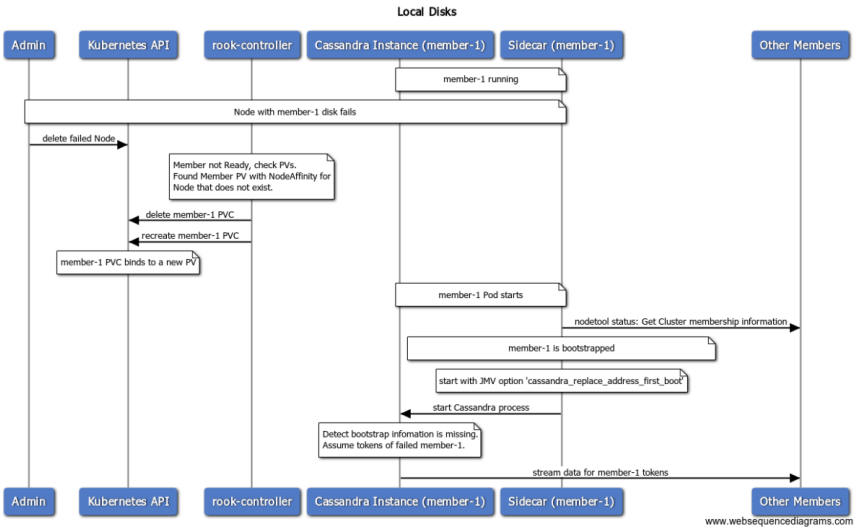


Figure 3.4: *Disk Loss Sequence Diagram*

Using this design, we can enable the use of local storage in many cloud environments.

4 Implementation

In this chapter, we give a detailed description of the development process for the Cassandra Operator, using the proposed best practices for developing Kubernetes Operators. We analyse the obstacles encountered while trying to put our design decisions into practice and the steps we took in order to overcome them. We describe methods and tools used to test and confirm the correct functionality of the Cassandra Operator. Finally, we discuss code refactorings used when developing the official Scylla Operator, a Kubernetes Operator for managing ScyllaDB.

4.1 Software Stack

4.1.1 Programming Language

One of the first decisions we had to make before we could start developing the Cassandra Operator, was to select a programming language to write it in. Theoretically, a Kubernetes Operator can be written in any language, as the only thing needed is a Kubernetes client to communicate with the Kubernetes API. Practically, the vast majority of open source Operators, as well as Kubernetes-related projects, are written in Go. [10]

Go is an open source, strongly typed programming language that focuses on concurrency. It features a syntax that is similar to C and boasts similar performance. All in all, Go is a popular language with no serious drawbacks for the specific application.

Choosing Go as our programming language provides the following advantages:

1. Many Operators and Kubernetes-related projects, that we could use to learn about developing software for Kubernetes, are written in Go. Kubernetes itself is written in Go.

2. Better libraries and frameworks for writing Kubernetes Operators.

3. A vibrant community and large growth over the last years.

4. Excellent documentation and good standard libraries.

As we saw, Go provides many advantages is overall a good fit for writing a Kubernetes Operator. Choosing a different language would not provide any clear advantages but would lack the support that Go enjoys from the Kubernetes community. Considering this, we make the decision to use Go for developing the Cassandra Operator.

4.1.2 Libraries and Frameworks

After choosing Go as our programming language, we look at libraries and frameworks that will help us develop a Kubernetes Operator. To explore such tools, we look at the source code of existing Kubernetes Operators. Navigator by Jetstack, [11] MySQL Operator by Oracle [12] and sample-controller in the official Kubernetes GitHub account [13] all use the client-go library [14] for communicating with the Kubernetes API. The client-go library includes several performance enhancements to the basic client library, mainly client-side caching of Kubernetes Objects. We will go into more details about the internals of client-go later in this chapter.

While searching for the existence of frameworks for writing Kubernetes Operators, we quickly realized that there were few solutions. At the time we started developing the Cassandra Operator, an up-and-coming Operator framework was operator-sdk by Red Hat. It was a promising project with a big company behind it, but it lacked maturity, adoption and breaking changes to the API happened very often. Because of this, it was decided that the operator-sdk was not a good fit for our development needs. In hindsight, it was a good decision as the operator-sdk underwent major refactors that would require us to constantly struggle in order to keep up.

Finally, we came upon the Rook project, which claimed to offer a framework for integrating storage providers with Kubernetes. Seeing that Rook is a very successful project and its contributors have a lot of experience working with stateful systems on Kubernetes, it immediately picked our interest. We explored its codebase and the integration of CockroachDB [15] to examine the benefits Rook had to offer.

After careful consideration, we decided to leverage the Rook framework, because of the following advantages:

- A set of common guidelines for CRDs to follow, in order to ensure uniformity across Operators of different storage providers.
- A complete and comprehensive testing framework, integrated with Jenkins.
- A nice build system, without external dependencies and based on make, enables the developer to quickly get started building an Operator.
- A vibrant community that will test and use Cassandra Operator.
- Our work will be peer reviewed by industry experts with experience on integrating storage systems with Kubernetes.

Summing up the above sections, our technology stack is comprised of:

1. *Go* as the programming language.
2. *client-go* as the Kubernetes API client library.
3. *Rook* as the framework to write our Operator in.

4.2 Implementation Architecture

Having selected our software stack, it is now time to plan the code architecture of our implementation. To get an idea of the best practices for writing a custom Controller, we look to the sample-controller [13] repository, which includes simple custom Controller using the best practices.

First, we need to explore the mechanisms that client-go offers and how we will leverage them in a custom Controller. The following is a pictorial representation of the client-go components and how they are used by a custom Controller.

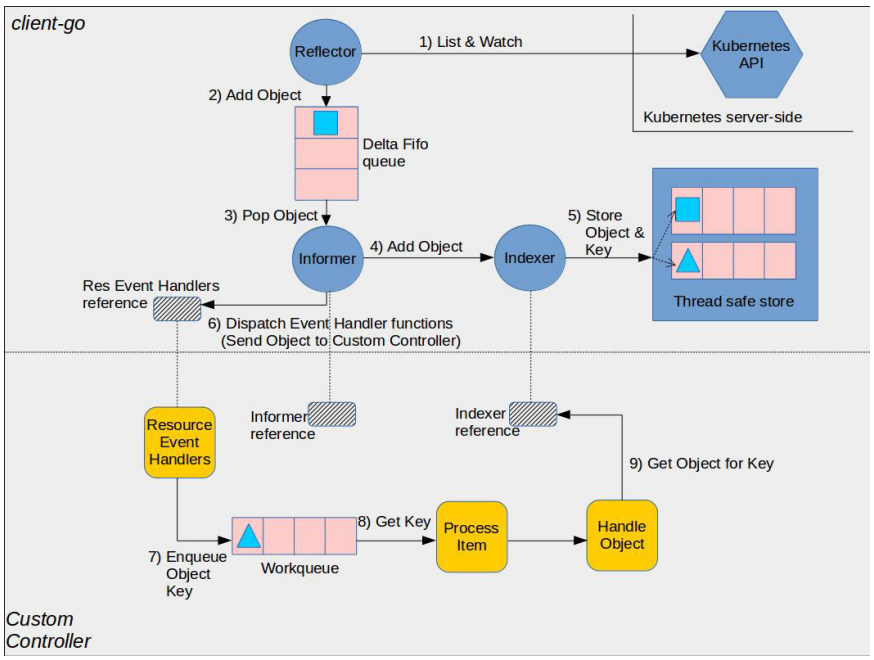


Figure 4.1: *client-go components, image from sample-controller repository*

4.2.1 client-go Components

- **Reflector**: The Reflector talks to the Kubernetes API. It initially lists every Object for a particular resource (eg Pods) and then watches for changes to Objects of that kind. It reflects changes to a Store, which in our case is the DeltaFIFO queue. The DeltaFifo queue stores Deltas, which are structs containing the new Object and the type of change that happened (Added, Updated or Deleted). [1]

- **Informer**: The Informer pops items from the DeltaFIFO queue and saves it for later retrieval. It usually uses an Indexer, for high performance retrieval. Most importantly, the Informer provides hooks called EventHandlers where a developer can run a custom function every time an Object is Added, Updated or Deleted.

[1] `https://github.com/kubernetes/client-go/blob/5b8ea8e61cb840f5c6f6baacd67ae93fe30a3027/tools/cache/delta_fifo.go#L596-L615`

- **Indexer**: The Indexer provides indexing functionality over the Objects of a Store. A typical indexing use-case is to create an index based on object labels. Indexer can maintain indexes based on several indexing functions. Indexer uses a thread-safe data store to store objects and their keys. A default function of Indexer is MetaNamespaceKeyFunc, which generates an object's key as "<namespace>/<name>".

4.2.2 Idiomatic Controller Components

From the many details of the client-go components, we focus on two main points. Using client-go informers, we can:

1. Register our own functions to run whenever an Object resource we watch is changed (Added, Updated, Deleted).

2. Read Objects from a local Store efficiently using Indexers, which prevents expensive network traffic.

With these building blocks, we go on to describe the architecture of a sample controller:

- **Resource Event Handlers**: These are the callback functions that Informer provides. The idiomatic pattern is to read the changed Object's MetaNamespaceKey, formed as "<namespace>/<name>", and enqueue that key in a Workqueue for further processing.

- **Workqueue**: This is a queue we create in our controller code, in order to decouple delivery of an object from its processing. As mentioned, resource event handler functions are written to extract the delivered object's key and add that to the work queue.

- **Process Item**: This is the function that pops items from the Workqueue and process them. Remember that the item we extract from the Workqueue is simply a key ("<namespace>/<name>"). After extracting it, we can retrieve the actual Object from the Store using an Indexer.

4.2.3 Summary of the Workflow

Having seen all components in detail, let's sum up the workflow our code is going to follow:

1. Create an Informer for each Object kind of interest and a Workqueue.
2. Setup the Cassandra Cluster Informer's Resource Event Handler callback funcs to extract the Object key ("<namespace>/<name>") and push it in the Workqueue.
3. A sync function will pop items from the Workqueue, get the Cassandra Cluster Object, that has changed, through the Indexer and proceed with our custom process logic.

4.3 Cluster Controller Scaffolding

4.3.1 Cluster Object Definition

In the previous section, we laid out a plan for the general workflow of our controller. Based on that and on the code of sample-controller, we create the scaffolding for our controller, the general structure without our custom processing code.

First of all, we have to define the struct that will hold our Cassandra Cluster Objects, as we described it earlier in the Design chapter.

```
package v1alpha1

import (
    rook "github.com/rook/rook/pkg/apis/rook.io/v1alpha2"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

const (
```

```
    APIVersion = CustomResourceGroup + "/" + Version
)

//
↪  ************************************************************************
// IMPORTANT FOR CODE GENERATION
// If the types in this file are updated, you will need to run
// `make codegen` to generate the new types under the client/clientset
↪  folder.
//
↪  ************************************************************************

// Kubernetes API Conventions:
//
↪  https://github.com/kubernetes/community/blob/af5c40530f50c3b36c13438187b311102093e
// Applicable Here:
//  * Optional fields use a pointer to correctly handle empty values.

// +genclient
// +genclient:noStatus
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type Cluster struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata"`
    Spec              ClusterSpec   `json:"spec"`
    Status            ClusterStatus `json:"status"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type ClusterList struct {
    metav1.TypeMeta `json:",inline"`
```

```
    metav1.ListMeta `json:"metadata"`
    Items           []Cluster `json:"items"`
}

// ClusterSpec is the desired state for a Cassandra Cluster.
type ClusterSpec struct {
    // Version of Cassandra to use.
    Version string `json:"version"`
    // Repository to pull the image from.
    Repository *string `json:"repository,omitempty"`
    // Mode selects an operating mode.
    Mode ClusterMode `json:"mode,omitempty"`
    // Datacenter that will make up this cluster.
    Datacenter DatacenterSpec `json:"datacenter"`
    // User-provided image for the sidecar that replaces default.
    SidecarImage *ImageSpec `json:"sidecarImage,omitempty"`
}

type ClusterMode string

const (
    ClusterModeCassandra ClusterMode = "cassandra"
    ClusterModeScylla    ClusterMode = "scylla"
)

// DatacenterSpec is the desired state for a Cassandra Datacenter.
type DatacenterSpec struct {
    // Name of the Cassandra Datacenter. Used in the
    ↪  cassandra-rackdc.properties file.
    Name string `json:"name"`
    // Racks of the specific Datacenter.
    Racks []RackSpec `json:"racks"`
}
```

```
// RackSpec is the desired state for a Cassandra Rack.
type RackSpec struct {
    // Name of the Cassandra Rack. Used in the
    ↪  cassandra-rackdc.properties file.
    Name string `json:"name"`
    // Members is the number of Cassandra instances in this rack.
    Members int32 `json:"members"`
    // User-provided ConfigMap applied to the specific statefulset.
    ConfigMapName *string `json:"configMapName,omitempty"`
    // Storage describes the underlying storage that Cassandra will
    ↪  consume.
    Storage rook.StorageScopeSpec `json:"storage"`
    // Placement describes restrictions for the nodes Cassandra is
    ↪  scheduled on.
    Placement *rook.Placement `json:"placement,omitempty"`
    // Resources the Cassandra Pods will use.
    Resources corev1.ResourceRequirements `json:"resources"`
}

// ImageSpec is the desired state for a container image.
type ImageSpec struct {
    // Version of the image.
    Version string `json:"version"`
    // Repository to pull the image from.
    Repository string `json:"repository"`
}

// ClusterStatus is the status of a Cassandra Cluster
type ClusterStatus struct {
    Racks map[string]*RackStatus `json:"racks,omitempty"`
}
```

```
// RackStatus is the status of a Cassandra Rack
type RackStatus struct {
    // Members is the current number of members requested in the
    ↪  specific Rack
    Members int32 `json:"members"`
    // ReadyMembers is the number of ready members in the specific Rack
    ReadyMembers int32 `json:"readyMembers"`
}
```

This code must be put inside the ”pkg/apis/cassandra.rook.io/v1alpha1/types.go” file. The ”pkg/apis/cassandra.rook.io” and ”pkg/apis/cassandra.rook.io/v1alpha1” packages contain some boilerplate files that are standard and won't be included here for brevity.

As we can see, the fields of the structs contain json tags, which specify the name of the field when they get unmarshalled to json text. In addition, there are some special comments like ”// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object” and ”// +genClient”. These comments are commands for a code-generator [16] that will parse our struct and generate a client and an informer for our custom Objects.

4.3.2 Entrypoint Command

After specifying our custom Object's structure, we need to create Informers for the Object kinds we are going to keep watch and keep local caches for. We use the factory methods, provided by client-go, to create SharedInformers, that is Informers that can share their cache with multiple controllers.

```
import (
    "github.com/rook/rook/cmd/rook/rook"
    rookinformers
    ↪  "github.com/rook/rook/pkg/client/informers/externalversions"
    kubeinformers "k8s.io/client-go/informers"
    "github.com/rook/rook/pkg/util/flags"
```

```
    "github.com/spf13/cobra"
    ...
)

const resyncPeriod = time.Second * 30

var operatorCmd = &cobra.Command{
    Use:   "operator",
    Short: "Runs the cassandra operator to deploy and manage cassandra
    ↪  in Kubernetes",
    Long: `Runs the cassandra operator to deploy and manage cassandra
    ↪  in kubernetes clusters.
https://github.com/rook/rook`,
}

func init() {
    flags.SetFlagsFromEnv(operatorCmd.Flags(), rook.RookEnvVarPrefix)

    operatorCmd.RunE = startOperator
}

func startOperator(cmd *cobra.Command, args []string) error {
...

    kubeClient, _, rookClient, err := rook.GetClientset()
    if err != nil {
        rook.TerminateFatal(fmt.Errorf("failed to get k8s clients.
        ↪  %+v\n", err))
    }

    kubeInformerFactory :=
    ↪  kubeinformers.NewSharedInformerFactory(kubeClient,
    ↪  resyncPeriod)
```

```
rookInformerFactory :=
↪ rookinformers.NewSharedInformerFactory(rookClient,
↪ resyncPeriod)
```

Every operator in Rook is essentially a CLI application, made with a CLI library called cobra. This is what the ”var operatorCmd” and ”init()” func are for. Rook is compiled into a single binary which includes all operators, available with different subcommands. [17] Our custom Controller is executed by running ”rook cassandra operator”.

The kubeInformerFactory creates Informers for the Object kinds included with Kubernetes. The rookInformerFactory creates Informers for the custom Object kinds (CRDs) introduced by rook. The rookinformers package (”pkg/client/informers/externalversions”) is actually generated by the code-generator we mentioned.

Now that we have created the Informer factories, we can initiate our custom Controller and pass it the Informers it needs.

```
c := controller.New(
    rookImage,
    kubeClient,
    rookClient,
    rookInformerFactory.Cassandra().V1alpha1().Clusters(),
    kubeInformerFactory.Apps().V1().StatefulSets(),
    kubeInformerFactory.Core().V1().Services(),
    kubeInformerFactory.Core().V1().Pods(),
    kubeInformerFactory.Core().V1().ServiceAccounts(),
    kubeInformerFactory.Rbac().V1().Roles(),
    kubeInformerFactory.Rbac().V1().RoleBindings(),
)

// Create a channel to receive OS signals
stopCh := server.SetupSignalHandler()
```

```
    // Start the informer factories
    go kubeInformerFactory.Start(stopCh)
    go rookInformerFactory.Start(stopCh)

    // Start the controller
    if err = c.Run(1, stopCh); err != nil {
        logger.Fatalf("Error running controller: %s", err.Error())
    }

    return nil
}
```

We create a new instance of the ClusterController struct, start the Informer factories in new goroutines and finally start running the controller. All of this code is included in the ”cmd/rook/cassandra/operator.go” file.

4.3.3 Reconciliation Loop

In the previous section, we wrote the code for registering our ”rook cassandra operator” command and creating the necessary Informers. Now we start building the controller. The package we will use is ”pkg/operator/cassandra/controller” and the file containing the basic scaffolding for the controller is ”pkg/operator/cassandra/-controller/controller.go”.

Let’s take a look at the controller.New function we saw earlier:

```
// ClusterController encapsulates all the tools the controller needs
// in order to talk to the Kubernetes API
type ClusterController struct {
    rookImage               string
    kubeClient              kubernetes.Interface
    rookClient              rookClientset.Interface
    clusterLister           listersv1alpha1.ClusterLister
```

```
    clusterListerSynced     cache.InformerSynced
    statefulSetLister       appslisters.StatefulSetLister
    statefulSetListerSynced cache.InformerSynced
    serviceLister           corelisters.ServiceLister
    serviceListerSynced     cache.InformerSynced
    podLister               corelisters.PodLister
    podListerSynced         cache.InformerSynced

    // queue is a rate limited work queue. This is used to queue work
    ↪  to be
    // processed instead of performing it as soon as a change happens.
    ↪  This
    // means we can ensure we only process a fixed amount of resources
    ↪  at a
    // time, and makes it easy to ensure we are never processing the
    ↪  same item
    // simultaneously in two different workers.
    queue workqueue.RateLimitingInterface
    // recorder is an event recorder for recording Event resources to
    ↪  the Kubernetes API
    recorder record.EventRecorder
}

// New returns a new ClusterController
func New(
    rookImage string,
    kubeClient kubernetes.Interface,
    rookClient rookClientset.Interface,
    clusterInformer informersv1alpha1.ClusterInformer,
    statefulSetInformer appsinformers.StatefulSetInformer,
    serviceInformer coreinformers.ServiceInformer,
    podInformer coreinformers.PodInformer,
    serviceAccountInformer coreinformers.ServiceAccountInformer,
```

```
    roleInformer rbacinformers.RoleInformer,
    roleBindingInformer rbacinformers.RoleBindingInformer,
) *ClusterController {

    // Add sample-controller types to the default Kubernetes Scheme so
    ↪  Events can be
    // logged for sample-controller types.
    rookScheme.AddToScheme(scheme.Scheme)
    // Create event broadcaster
    logger.Infof("creating event broadcaster...")
    eventBroadcaster := record.NewBroadcaster()
    eventBroadcaster.StartLogging(logger.Infof)
    eventBroadcaster.StartRecordingToSink(&typedcorev1.EventSinkImpl{Interface:
    ↪  kubeClient.CoreV1().Events("")})
    recorder := eventBroadcaster.NewRecorder(scheme.Scheme,
    ↪  corev1.EventSource{Component: controllerName})

    cc := &ClusterController{
        rookImage:  rookImage,
        kubeClient: kubeClient,
        rookClient: rookClient,

        clusterLister:           clusterInformer.Lister(),
        clusterListerSynced:     clusterInformer.Informer().HasSynced,
        statefulSetLister:       statefulSetInformer.Lister(),
        statefulSetListerSynced:
        ↪  statefulSetInformer.Informer().HasSynced,
        podLister:               podInformer.Lister(),
        podListerSynced:         podInformer.Informer().HasSynced,
        serviceLister:           serviceInformer.Lister(),
        serviceListerSynced:     serviceInformer.Informer().HasSynced,
```

```
        queue:
        ↪  workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(),
        ↪  clusterQueueName),
        recorder: recorder,
    }

    // Add event handling functions

    clusterInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            newCluster := obj.(*cassandrav1alpha1.Cluster)
            cc.enqueueCluster(newCluster)
        },
        UpdateFunc: func(old, new interface{}) {
            newCluster := new.(*cassandrav1alpha1.Cluster)
            oldCluster := old.(*cassandrav1alpha1.Cluster)
            // If the Spec is the same as the one in our cache, there
            ↪  aren't
            // any changes we are interested in.
            if reflect.DeepEqual(newCluster.Spec, oldCluster.Spec) {
                return
            }
            cc.enqueueCluster(newCluster)
        },
        DeleteFunc: func(obj interface{}) {
            // TODO: handle deletion
        },
    })

    statefulSetInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: cc.handleObject,
        UpdateFunc: func(old, new interface{}) {
            newStatefulSet := new.(*appsv1.StatefulSet)
```

```
            oldStatefulSet := old.(*appsv1.StatefulSet)
            // If the StatefulSet is the same as the one in our cache,
            ↪  there
            // is no use adding it again.
            if newStatefulSet.ResourceVersion ==
            ↪  oldStatefulSet.ResourceVersion {
                return
            }
            // If ObservedGeneration != Generation, it means that the
            ↪  StatefulSet controller
            // has not yet processed the current StatefulSet object.
            // That means its Status is stale and we don't want to
            ↪  queue it.
            if newStatefulSet.Status.ObservedGeneration !=
            ↪  newStatefulSet.Generation {
                return
            }
            cc.handleObject(new)
        },
        DeleteFunc: cc.handleObject,
    })

    serviceInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            service := obj.(*corev1.Service)
            if service.Spec.ClusterIP == corev1.ClusterIPNone {
                return
            }
            cc.handleObject(obj)
        },
        UpdateFunc: func(old, new interface{}) {
            newService := new.(*corev1.Service)
            oldService := old.(*corev1.Service)
```

```
            if oldService.ResourceVersion == newService.ResourceVersion
            ↪ {
                return
            }
            cc.handleObject(new)
        },
        DeleteFunc: func(obj interface{}) {
            // TODO: investigate if further action needs to be taken
        },
    })

    return cc
}
```

The controller.New function returns a pointer to a ClusterController struct. First, a new ClusterController struct is created and it is filled with the Informers passed to the New function. Afterwards, we setup the Event Handler callback functions for the ClusterInformer, StatefulSetInformer and ServiceInformer.

Cassandra Clusters are enqueued in the workqueue by calling `cc.enqueueCluster`:

```
// enqueueCluster takes a Cluster resource and converts it into a
↪ namespace/name
// string which is then put onto the work queue. This method should not
↪ be
// passed resources of any type other than Cluster.
func (cc *ClusterController) enqueueCluster(obj
↪ *cassandrav1alpha1.Cluster) {
    var key string
    var err error
    if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
        runtime.HandleError(err)
        return
```

```
    }
    cc.queue.AddRateLimited(key)
}
```

We can clearly see the pattern we talked about earlier. Our Event Handler functions are being called by the ClusterInformer and are placing the Object's key ("<namespace>/<name>") in a workqueue.

For the StatefulSetInformer and ServiceInformer, we also want to sync a Cassandra Cluster when one of its created StatefulSets of Services changes. But how do we know which Cluster the Service or StatefulSet belongs to ? We simply look at the OwnerReferences field in the Object's metadata. We set this field when we create Objects for a Cluster and then we can use them to find out which Cluster the Object belongs to.

This is what the `cc.handleObject` function does. It takes an Object and checks if it belongs to a Cassandra Cluster. If it does, then it enqueues the corresponding Cluster Object key, using :

```
// handleObject will take any resource implementing metav1.Object and
↪  attempt
// to find the Cluster resource that 'owns' it. It does this by looking
↪  at the
// objects metadata.ownerReferences field for an appropriate
↪  OwnerReference.
// It then enqueues that Cluster resource to be processed. If the
↪  object does not
// have an appropriate OwnerReference, it will simply be skipped.
func (cc *ClusterController) handleObject(obj interface{}) {
    var object metav1.Object
    var ok bool
    if object, ok = obj.(metav1.Object); !ok {
        tombstone, ok := obj.(cache.DeletedFinalStateUnknown)
        if !ok {
```

```
            runtime.HandleError(fmt.Errorf("error decoding object,
            ↪ invalid type"))
            return
        }
        object, ok = tombstone.Obj.(metav1.Object)
        if !ok {
            runtime.HandleError(fmt.Errorf("error decoding object
            ↪ tombstone, invalid type"))
            return
        }
        logger.Infof("Recovered deleted object '%s' from tombstone",
        ↪ object.GetName())
    }
    logger.Infof("Processing object: %s", object.GetName())
    if ownerRef := metav1.GetControllerOf(object); ownerRef != nil {
        // If the object is not a Cluster or doesn't belong to our
        ↪ APIVersion, skip it.
        if ownerRef.Kind != "Cluster" || ownerRef.APIVersion !=
        ↪ cassandrav1alpha1.APIVersion {
            return
        }

        cluster, err :=
        ↪ cc.clusterLister.Clusters(object.GetNamespace()).Get(ownerRef.Name)
        if err != nil {
            logger.Infof("ignoring orphaned object '%s' of cluster
            ↪ '%s'", object.GetSelfLink(), ownerRef.Name)
            return
        }

        cc.enqueueCluster(cluster)
        return
    }
```

```
}
```

We have completed the first two steps of the Cluster Controller workflow. Informers watch Objects and call our Event Handler functions when a change happens. Event Handler functions then enqueue the associated Cluster Object in the workqueue. Now, we move to the processing function, which will pop items from the workqueue and process them.

This whole procedure is started with the Run function of the ClusterController:

```
// Run starts the ClusterController process loop
func (cc *ClusterController) Run(threadiness int, stopCh <-chan
↪  struct{}) error {
    defer runtime.HandleCrash()
    defer cc.queue.ShutDown()

    //     Start the informer factories to begin populating the
    ↪  informer caches
    logger.Info("starting cassandra controller")

    // Wait for the caches to be synced before starting workers
    logger.Info("waiting for informers caches to sync...")
    if ok := cache.WaitForCacheSync(
        stopCh,
        cc.clusterListerSynced,
        cc.statefulSetListerSynced,
        cc.podListerSynced,
        cc.serviceListerSynced,
    ); !ok {
        return fmt.Errorf("failed to wait for caches to sync")
    }

    logger.Info("starting workers")
```

```
    for i := 0; i < threadiness; i++ {
        go wait.Until(cc.runWorker, time.Second, stopCh)
    }

    logger.Info("started workers")
    <-stopCh
    logger.Info("Shutting down cassandra controller workers")

    return nil
}

func (cc *ClusterController) runWorker() {
    for cc.processNextWorkItem() {
    }
}

func (cc *ClusterController) processNextWorkItem() bool {
    obj, shutdown := cc.queue.Get()

    if shutdown {
        return false
    }

    err := func(obj interface{}) error {
        defer cc.queue.Done(obj)
        key, ok := obj.(string)
        if !ok {
            cc.queue.Forget(obj)
            runtime.HandleError(fmt.Errorf("expected string in queue
            ↪  but got %#v", obj))
        }
        if err := cc.syncHandler(key); err != nil {
            cc.queue.AddRateLimited(key)
```

```
            return fmt.Errorf("error syncing '%s', requeueing: %s",
            ↪  key, err.Error())
        }
        cc.queue.Forget(obj)
        logger.Infof("Successfully synced '%s'", key)
        return nil
    }(obj)

    if err != nil {
        runtime.HandleError(err)
        return true
    }

    return true
}

// syncHandler compares the actual state with the desired, and attempts
↪  to
// converge the two. It then updates the Status block of the Cluster
// resource with the current status of the resource.
func (cc *ClusterController) syncHandler(key string) error {

    // Convert the namespace/name string into a distinct namespace and
    ↪  name.
    namespace, name, err := cache.SplitMetaNamespaceKey(key)
    if err != nil {
        runtime.HandleError(fmt.Errorf("invalid resource key: %s",
        ↪  key))
        return nil
    }

    // Get the Cluster resource with this namespace/name
    cluster, err := cc.clusterLister.Clusters(namespace).Get(name)
```

```
    if err != nil {
        // The Cluster resource may no longer exist, in which case we
        ↪  stop processing.
        if apierrors.IsNotFound(err) {
            runtime.HandleError(fmt.Errorf("cluster '%s' in work queue
            ↪  no longer exists", key))
            return nil
        }
        return fmt.Errorf("Unexpected error while getting cluster
        ↪  object: %s", err)
    }

    logger.Infof("handling cluster object: %+v", spew.Sdump(cluster))
    // Deepcopy here to ensure nobody messes with the cache.
    old, new := cluster, cluster.DeepCopy()
    // If sync was successful and Status has changed, update the
    ↪  Cluster.
    if err = cc.Sync(new); err == nil && !reflect.DeepEqual(old.Status,
    ↪  new.Status) {
        err = util.PatchClusterStatus(new, cc.rookClient)
    }

    return err
}
```

The `Run` method waits for the Informers' caches to sync for the first time and then starts the workers which will pop items from the workqueue and process them.

Each worker then enters an infinite loop processing items with the `syncHandler` function. Inside `syncHandler`, the Cluster Object is retrieved (`cluster, err := cc.clusterLister.Clusters(namespace).Get(name)`) and the `Sync` function is called, which contains the logic of our Controller. After that, we check if the Cluster Status changed during the sync. If it has, we update the Cluster Object in the API Server.

This concludes the basic scaffolding of the Cluster Controller, based on the sample-controller [13] repository.

4.4 Build

Having our basic scaffolding ready, we can proceed to define the image for our Operator. The image is defined by the following Dockerfile:

```
FROM alpine:3.8

ARG ARCH
ARG TINI_VERSION

ADD rook /usr/local/bin/

# Add files for the sidecar
RUN mkdir -p /sidecar
RUN mkdir -p /sidecar/plugins

ADD rook /sidecar/
# Jolokia plugin for JMX<->HTTP
ADD
↪  "http://search.maven.org/remotecontent?filepath=org/jolokia/jolokia-jvm/1.6.0/jolol
↪  /sidecar/plugins/jolokia.jar

# Run tini as PID 1 and avoid signal handling issues
ADD
↪  https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini-static-${ARC
↪  /sidecar/tini
RUN chmod +x /sidecar/tini && chmod +x /usr/local/bin/rook


ENTRYPOINT ["/sidecar/tini", "--", "/usr/local/bin/rook"]
```

```
CMD [""]
```

The image includes the statically compiled rook binary, the jolokia javaagent for the sidecar and tini [2], a minimal init to avoid running our process as PID 1.

The same image is used for both the Cluster Controller and the Sidecar, albeit with different arguments. The Cluster Controller is started with ”rook cassandra operator” while the Sidecar is started with ”rook cassandra sidecar”.

The image for the Cassandra Operator can be built using `make images="cassandra" build`

4.5 Deployment

Having the Docker image of the Operator, we can now deploy it. The Cluster Controller will run in a StatefulSet with 1 replica. We choose a StatefulSet instead of a Deployment, because they guarantee that at most 1 Pod will ever exist for each replica.

First, we create the CRD Object to register our custom endpoint:

cluster/examples/kubernetes/cassandra/operator.yaml

```
\begin{minted}[breaklines,label=cluster/examples/kubernetes/cassandra/operator.yaml]{yaml}
# Cassandra Cluster CRD
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: clusters.cassandra.rook.io
spec:
  group: cassandra.rook.io
  names:
    kind: Cluster
    listKind: ClusterList
    plural: clusters
    singular: cluster
```

[2] tini https://github.com/krallin/tini

```
  scope: Namespaced
  version: v1alpha1
  validation:
    openAPIV3Schema:
      properties:
        spec:
          type: object
          properties:
            version:
              type: string
              description: "Version of Cassandra"
            datacenter:
              type: object
              properties:
                name:
                  type: string
                  description: "Datacenter Name"
                racks:
                  type: array
                  properties:
                    name:
                      type: string
                    members:
                      type: integer
                    configMapName:
                      type: string
                    storage:
                      type: object
                      properties:
                        volumeClaimTemplates:
                          type: object
                          # TODO: Check if we can ref the already
                          ↪  existing schema
```

```
            required:
             - "volumeClaimTemplates"
          placement:
            type: object
          resources:
            type: object
            properties:
              # TODO: Check if we can ref the already
              ↪  existing schema
              cassandra:
                type: object
              sidecar:
                type: object
            required:
              - "cassandra"
              - "sidecar"
          sidecarImage:
            type: object
        required:
          - "name"
          - "members"
          - "storage"
          - "resources"
      required:
        - "name"
  required:
    - "version"
    - "datacenter"
```

After the CRD is created, we create the StatefulSet for the Cluster Controller along with the necessary RBAC objects, to give it the needed permissions:

cluster/examples/kubernetes/cassandra/operator.yaml

```
# ClusterRole for cassandra-operator.
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: rook-cassandra-operator
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - list
      - watch
      - delete
  - apiGroups:
      - ""
    resources:
      - services
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - persistentvolumes
      - persistentvolumeclaims
    verbs:
      - get
      - delete
  - apiGroups:
      - ""
    resources:
      - nodes
    verbs:
```

```
      - get
  - apiGroups:
      - apps
    resources:
      - statefulsets
    verbs:
      - "*"
  - apiGroups:
      - policy
    resources:
      - poddisruptionbudgets
    verbs:
      - create
  - apiGroups:
      - cassandra.rook.io
    resources:
      - "*"
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - create
      - update
      - patch
---
# ServiceAccount for cassandra-operator. Serves as its authorization
↪  identity.
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# Bind cassandra-operator ServiceAccount with ClusterRole.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rook-cassandra-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-cassandra-operator
subjects:
- kind: ServiceAccount
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# cassandra-operator StatefulSet.
 apiVersion: apps/v1
 kind: StatefulSet
 metadata:
   name: rook-cassandra-operator
   namespace: rook-cassandra-system
   labels:
     app: rook-cassandra-operator
 spec:
   replicas: 1
   serviceName: "non-existent-service"
   selector:
     matchLabels:
       app: rook-cassandra-operator
   template:
     metadata:
```

```
    labels:
      app: rook-cassandra-operator
  spec:
    serviceAccountName: rook-cassandra-operator
    containers:
    - name: rook-cassandra-operator
      image: rook/cassandra:master
      imagePullPolicy: "Always"
      args: ["cassandra", "operator"]
      env:
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
```

4.6 Reconciliation Logic

We have defined our Cluster Custom Resource in code and have set up a reconciliation loop that is executed each time a Cluster or Cluster-related Object changes.

Now that we have the Controller scaffolding ready, we can begin to write the code that will run on each Cluster reconciliation. The function implementing this is the `Sync` method of the ClusterController struct, inside ”pkg/operator/cassandra/controller/sync.go”

```
// Sync attempts to sync the given Cassandra Cluster.
// NOTE: the Cluster Object is a DeepCopy. Modify at will.
func (cc *ClusterController) Sync(c *cassandrav1alpha1.Cluster) error {
```

```
    // Sync Headless Service for Cluster
    if err := cc.syncClusterHeadlessService(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageHeadlessServiceSyncFailed,
        )
        return err
    }

    // Sync Cluster Member Services
    if err := cc.syncMemberServices(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageMemberServicesSyncFailed,
        )
        return err
    }

    // Update Status
    if err := cc.updateStatus(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageUpdateStatusFailed,
        )
        return err
    }
```

```
    // Sync Cluster
    if err := cc.syncCluster(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageClusterSyncFailed,
        )
        return err
    }

    return nil
}
```

As we can see, the `Sync` function reconciles a number of Objects. We will analyze how they relate to the action workflows we defined in the design chapter (3).

4.6.1 Cluster Creation & Scale Up

First of all, let's remember the design we created in subsection 3.9.1. The sequence diagram of the Cluster Creation and Scale Up workflow is the following:

Following the sequence diagram, our Controller needs to create a Headless Service for the Casasndra Cluster, which clients will use to connect to a healthy Cassandra instance. In addition, the Cluster Controller should create ClusterIP Services for existing Members of the Cluster, which we will call those MemberServices. This functionality is implemented with the `syncClusterHeadlessService` and `syncMemberServices` functions, called from the `Sync` method:

pkg/operator/cassandra/controller/service.go

```
// SyncClusterHeadlessService checks if a Headless Service exists
// for the given Cluster, in order for the StatefulSets to utilize it.
// If it doesn't exists, then create it.
```

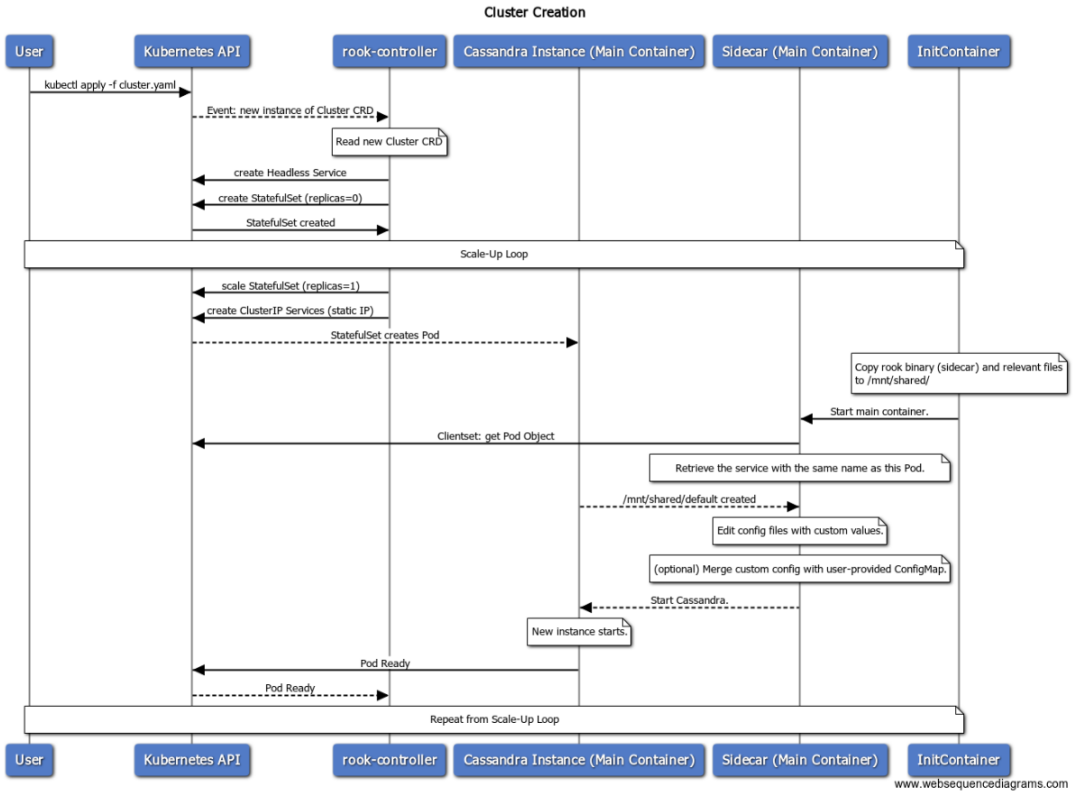


Figure 4.2: *Cluster Creation & Scale Up Sequence Diagram, as seen in subsection 3.9.1*

```
func (cc *ClusterController) syncClusterHeadlessService(c
↪  *cassandrav1alpha1.Cluster) error {
    clusterHeadlessService := &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:            util.HeadlessServiceNameForCluster(c),
            Namespace:       c.Namespace,
            Labels:          util.ClusterLabels(c),
            OwnerReferences:
             ↪  []metav1.OwnerReference{util.NewControllerRef(c)},
        },
        Spec: corev1.ServiceSpec{
            ClusterIP: corev1.ClusterIPNone,
            Type:      corev1.ServiceTypeClusterIP,
            Selector:  util.ClusterLabels(c),
            // Necessary to specify a Port to work correctly
            // https://github.com/kubernetes/kubernetes/issues/32796
```

```
            // TODO: find in what version this was fixed
            Ports: []corev1.ServicePort{
                {
                    Name: "prometheus",
                    Port: 9180,
                },
            },
        },
    }

    logger.Infof("Syncing ClusterHeadlessService `%s` for Cluster
    ↪  `%s`", clusterHeadlessService.Name, c.Name)

    return cc.syncService(clusterHeadlessService, c)
}

// SyncMemberServices checks, for every Pod of the Cluster that
// has been created, if a corresponding ClusterIP Service exists,
// which will serve as a static ip.
// If it doesn't exist, it creates it.
// It also assigns the first two members of each rack as seeds.
func (cc *ClusterController) syncMemberServices(c
↪  *cassandrav1alpha1.Cluster) error {

    pods, err := util.GetPodsForCluster(c, cc.podLister)
    if err != nil {
        return err
    }

    // For every Pod of the cluster that exists, check that a
    // a corresponding ClusterIP Service exists, and if it doesn't,
    // create it.
    logger.Infof("Syncing MemberServices for Cluster `%s`", c.Name)
```

```
    for _, pod := range pods {
        if err := cc.syncService(memberServiceForPod(pod, c), c); err
        ↪  != nil {
            logger.Errorf("Error syncing member service for '%s'",
            ↪  pod.Name)
            return err
        }
    }
    return nil
}

// syncService checks if the given Service exists and creates it if it
↪  doesn't
// it creates it
func (cc *ClusterController) syncService(s *corev1.Service, c
↪  *cassandrav1alpha1.Cluster) error {

    existingService, err :=
    ↪  cc.serviceLister.Services(s.Namespace).Get(s.Name)
    // If we get an error but without the IsNotFound error raised
    // then something is wrong with the network, so requeue.
    if err != nil && !apierrors.IsNotFound(err) {
        return err
    }
    // If the service already exists, check that it's
    // controlled by the given Cluster
    if err == nil {
        return util.VerifyOwner(existingService, c)
    }

    // At this point, the Service doesn't exist, so we are free to
    ↪  create it
    _, err = cc.kubeClient.CoreV1().Services(s.Namespace).Create(s)
```

```
    return err

}

func memberServiceForPod(pod *corev1.Pod, cluster
↪  *cassandrav1alpha1.Cluster) *corev1.Service {

    labels := util.ClusterLabels(cluster)
    labels[constants.DatacenterNameLabel] =
    ↪  pod.Labels[constants.DatacenterNameLabel]
    labels[constants.RackNameLabel] =
    ↪  pod.Labels[constants.RackNameLabel]
    // If Member is seed, add the appropriate label
    if strings.HasSuffix(pod.Name, "-0") || strings.HasSuffix(pod.Name,
    ↪  "-1") {
        labels[constants.SeedLabel] = ""
    }

    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:            pod.Name,
            Namespace:       pod.Namespace,
            OwnerReferences:
            ↪  []metav1.OwnerReference{util.NewControllerRef(cluster)},
            Labels:          labels,
            Annotations:
            ↪  map[string]string{endpoint.TolerateUnreadyEndpointsAnnotation:
            ↪  "true"},
        },
        Spec: corev1.ServiceSpec{
            Type:     corev1.ServiceTypeClusterIP,
            Selector: util.StatefulSetPodLabel(pod.Name),
            Ports: []corev1.ServicePort{
```

```
                {
                    Name: "inter-node-communication",
                    Port: 7000,
                },
                {
                    Name: "ssl-inter-node-communication",
                    Port: 7001,
                },
                {
                    Name: "jmx-monitoring",
                    Port: 7199,
                },
                {
                    Name: "cql",
                    Port: 9042,
                },
                {
                    Name: "thrift",
                    Port: 9160,
                },
                {
                    Name: "cql-ssl",
                    Port: 9142,
                },
            },
            PublishNotReadyAddresses: true,
        },
    }
}
```

After syncing the Headless Service and MemberServices, we update the Cluster Status in-memory.

pkg/operator/cassandra/controller/cluster.go

```
// UpdateStatus updates the status of the given Cassandra Cluster.
// It doesn't post the result to the API Server yet.
// That will be done at the end of the sync loop.
func (cc *ClusterController) updateStatus(c *cassandrav1alpha1.Cluster)
↪  error {

    clusterStatus := cassandrav1alpha1.ClusterStatus{
        Racks: map[string]*cassandrav1alpha1.RackStatus{},
    }
    logger.Infof("Updating Status for cluster %s in namespace %s",
    ↪  c.Name, c.Namespace)

    for _, rack := range c.Spec.Datacenter.Racks {

        status := &cassandrav1alpha1.RackStatus{}

        // Get corresponding StatefulSet from lister
        sts, err := cc.statefulSetLister.StatefulSets(c.Namespace).
            Get(util.StatefulSetNameForRack(rack, c))
        // If it wasn't found, continue
        if apierrors.IsNotFound(err) {
            continue
        }
        // If we got a different error, requeue and log it
        if err != nil {
            return fmt.Errorf("error trying to get StatefulSet %s in
            ↪  namespace %s: %s", sts.Name, sts.Namespace,
            ↪  err.Error())
        }

        // Update Members
        status.Members = *sts.Spec.Replicas
```

```
        // Update ReadyMembers
        status.ReadyMembers = sts.Status.ReadyReplicas

        // Update Status for Rack
        clusterStatus.Racks[rack.Name] = status
    }

    c.Status = clusterStatus
    return nil
}
```

We have the Cluster Spec, written by the user. We also have the actual Cluster Status, which we just calculated. The next step is to compare the Cluster Status with the Cluster Spec and take any necessary actions to reconcile them. This is what the `syncCluster` method does:

pkg/operator/cassandra/controller/cluster.go

```
// SyncCluster checks the Status and performs reconciliation for
// the given Cassandra Cluster.
func (cc *ClusterController) syncCluster(c *cassandrav1alpha1.Cluster)
↪   error {

    // Check if any rack isn't created
    for _, rack := range c.Spec.Datacenter.Racks {
        // For each rack, check if a status entry exists
        if _, ok := c.Status.Racks[rack.Name]; !ok {
            logger.Infof("Attempting to create Rack %s", rack.Name)
            err := cc.createRack(rack, c)
            return err
        }
    }

    // Check that all racks are ready before taking any action
```

```
    for _, rack := range c.Spec.Datacenter.Racks {
        rackStatus := c.Status.Racks[rack.Name]
        if rackStatus.Members != rackStatus.ReadyMembers {
            logger.Infof("Rack %s is not ready, %+v", rack.Name,
            ↪  *rackStatus)
            return nil
        }
    }

    // Check if any rack needs to scale up
    for _, rack := range c.Spec.Datacenter.Racks {

        if rack.Members > c.Status.Racks[rack.Name].Members {
            logger.Infof("Attempting to scale rack %s", rack.Name)
            err := cc.scaleUpRack(rack, c)
            return err
        }
    }

    return nil
}
```

As shown in the sequence diagram, the Controller takes one action on each sync loop. First it checks if all the Racks defined in the Spec exist. If one doesn't, then it creates a new StatefulSet with 0 replicas for that Rack:

pkg/operator/cassandra/controller/cluster.go

```
// createRack creates a new Cassandra Rack with 0 Members.
func (cc *ClusterController) createRack(r cassandrav1alpha1.RackSpec, c
↪  *cassandrav1alpha1.Cluster) error {

    sts := util.StatefulSetForRack(r, c, cc.rookImage)
    existingStatefulset, err :=
    ↪  cc.statefulSetLister.StatefulSets(sts.Namespace).Get(sts.Name)
```

```
    if err == nil {
        return util.VerifyOwner(existingStatefulset, c)
    }
    if err != nil && !apierrors.IsNotFound(err) {
        return fmt.Errorf("Error trying to create StatefulSet %s in
        ↪  namespace %s : %s", sts.Name, sts.Namespace, err.Error())
    }

    _, err =
    ↪  cc.kubeClient.AppsV1().StatefulSets(sts.Namespace).Create(sts)

    if err == nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackCreated, r.Name),
        )
    }

    if err != nil {
        logger.Errorf("Unexpected error while creating rack for cluster
        ↪  %+v: %s", c, err.Error())
    }

    return err
}
```

The StatefulSet specification is defined in the ”pkg/operator/cassandra/controller/util/resource.go” file:

pkg/operator/cassandra/controller/util/resource.go

```
func StatefulSetForRack(r cassandrav1alpha1.RackSpec, c
↪  *cassandrav1alpha1.Cluster, rookImage string) *appsv1.StatefulSet {
```

```
rackLabels := RackLabels(r, c)
stsName := StatefulSetNameForRack(r, c)

return &appsv1.StatefulSet{
    ObjectMeta: metav1.ObjectMeta{
        Name:            stsName,
        Namespace:       c.Namespace,
        Labels:          rackLabels,
        OwnerReferences:
        ↪  []metav1.OwnerReference{NewControllerRef(c)},
    },
    Spec: appsv1.StatefulSetSpec{
        Replicas: RefFromInt32(0),
        // Use a common Headless Service for all StatefulSets
        ServiceName: HeadlessServiceNameForCluster(c),
        Selector: &metav1.LabelSelector{
            MatchLabels: rackLabels,
        },
        PodManagementPolicy: appsv1.OrderedReadyPodManagement,
        UpdateStrategy: appsv1.StatefulSetUpdateStrategy{
            Type: appsv1.RollingUpdateStatefulSetStrategyType,
        },
        Template: corev1.PodTemplateSpec{
            ObjectMeta: metav1.ObjectMeta{
                Labels: rackLabels,
            },
            Spec: corev1.PodSpec{
                Volumes: []corev1.Volume{
                    {
                        Name: "shared",
                        VolumeSource: corev1.VolumeSource{
```

```
                EmptyDir:
                ↪ &corev1.EmptyDirVolumeSource{},
            },
        },
    },
    InitContainers: []corev1.Container{
        {
            Name:            "rook-install",
            Image:           rookImage,
            ImagePullPolicy: "IfNotPresent",
            Command: []string{
                "/bin/sh",
                "-c",
                fmt.Sprintf("cp -a /sidecar/* %s",
                ↪ constants.SharedDirName),
            },
            VolumeMounts: []corev1.VolumeMount{
                {
                    Name:      "shared",
                    MountPath: constants.SharedDirName,
                    ReadOnly:  false,
                },
            },
        },
    },
    Containers: []corev1.Container{
        {
            Name:            "cassandra",
            Image:           ImageForCluster(c),
            ImagePullPolicy: "IfNotPresent",
            Ports: []corev1.ContainerPort{
                {
                    Name:          "intra-node",
```

```
                ContainerPort: 7000,
            },
            {
                Name:          "tls-intra-node",
                ContainerPort: 7001,
            },
            {
                Name:          "jmx",
                ContainerPort: 7199,
            },
            {
                Name:          "cql",
                ContainerPort: 9042,
            },
            {
                Name:          "thrift",
                ContainerPort: 9160,
            },
            {
                Name:          "jolokia",
                ContainerPort: 8778,
            },
            {
                Name:          "prometheus",
                ContainerPort: 9180,
            },
        },
        // TODO: unprivileged entrypoint
        Command: []string{
            fmt.Sprintf("%s/tini",
            ↪  constants.SharedDirName),
            "--",
```

```
            fmt.Sprintf("%s/rook",
            ↪  constants.SharedDirName),
        },
        Args: []string{
            "cassandra",
            "sidecar",
        },
        Env: []corev1.EnvVar{
            {
                Name: constants.PodIPEnvVar,
                ValueFrom: &corev1.EnvVarSource{
                    FieldRef:
                    ↪  &corev1.ObjectFieldSelector{
                        FieldPath: "status.podIP",
                    },
                },
            },
            {
                Name: k8sutil.PodNameEnvVar,
                ValueFrom: &corev1.EnvVarSource{
                    FieldRef:
                    ↪  &corev1.ObjectFieldSelector{
                        FieldPath: "metadata.name",
                    },
                },
            },
            {
                Name: k8sutil.PodNamespaceEnvVar,
                ValueFrom: &corev1.EnvVarSource{
                    FieldRef:
                    ↪  &corev1.ObjectFieldSelector{
                        FieldPath:
                        ↪  "metadata.namespace",
```

```
                    },
                },
            },
            {
                Name:
                ↪ constants.ResourceLimitCPUEnvVar,
                ValueFrom: &corev1.EnvVarSource{
                    ResourceFieldRef:
                    ↪ &corev1.ResourceFieldSelector{
                        ContainerName: "cassandra",
                        Resource:
                        ↪ "limits.cpu",
                        Divisor:
                        ↪ resource.MustParse("1"),
                    },
                },
            },
            {
                Name:
                ↪ constants.ResourceLimitMemoryEnvVar,
                ValueFrom: &corev1.EnvVarSource{
                    ResourceFieldRef:
                    ↪ &corev1.ResourceFieldSelector{
                        ContainerName: "cassandra",
                        Resource:
                        ↪ "limits.memory",
                        Divisor:
                        ↪ resource.MustParse("1Mi"),
                    },
                },
            },
        },
        Resources:    r.Resources,
```

```
VolumeMounts: volumeMountsForRack(r, c),
LivenessProbe: &corev1.Probe{
    // Initial delay should be big, because
    ↪  scylla runs benchmarks
    // to tune the IO settings.
    InitialDelaySeconds: int32(400),
    TimeoutSeconds:      int32(5),
    // TODO: Investigate if it's ok to call
    ↪  status every 10 seconds
    PeriodSeconds: int32(10),
    Handler: corev1.Handler{
        HTTPGet: &corev1.HTTPGetAction{
            Port:
            ↪  intstr.FromInt(constants.ProbePort),
            Path:
            ↪  constants.LivenessProbePath,
        },
    },
},
ReadinessProbe: &corev1.Probe{
    InitialDelaySeconds: int32(15),
    TimeoutSeconds:      int32(5),
    // TODO: Investigate if it's ok to call
    ↪  status every 10 seconds
    PeriodSeconds: int32(10),
    Handler: corev1.Handler{
        HTTPGet: &corev1.HTTPGetAction{
            Port:
            ↪  intstr.FromInt(constants.ProbePort),
            Path:
            ↪  constants.ReadinessProbePath,
        },
    },
```

```
                        },
                        // Before a Cassandra Pod is stopped,
                        ↪  execute nodetool drain to
                        // flush the memtable to disk and stop
                        ↪  listening for connections.
                        // This is necessary to ensure we don't
                        ↪  lose any data.
                        Lifecycle: &corev1.Lifecycle{
                            PreStop: &corev1.Handler{
                                Exec: &corev1.ExecAction{
                                    Command: []string{
                                        "nodetool",
                                        "drain",
                                    },
                                },
                            },
                        },
                    },
                },
                // Set GracePeriod to 2 days, should be enough even
                ↪  for the slowest of systems
                TerminationGracePeriodSeconds:
                ↪  RefFromInt64(200000),
                ServiceAccountName:
                ↪  ServiceAccountNameForMembers(c),
                Affinity:                    affinityForRack(r),
                Tolerations:
                ↪  tolerationsForRack(r),
            },
        },
        VolumeClaimTemplates:
        ↪  volumeClaimTemplatesForRack(r.Storage.VolumeClaimTemplates),
    },
```

```
    }
}
```

As we can see, the StatefulSet Pods include an InitContainer that will copy the rook binary to a shared volume. Then the Cassandra Pod will start and execute the rook binary from the shared volume.

After that, we check that all Racks are in good condition, meaning all of their Members are Running and Ready. This is because we don't want to initiate an operational action if the Cluster is not in good health.

Having confirmed that all Racks are in good health, we compare the requested Members with the actual Members for every Rack to see if any Rack needs to scale up:

pkg/operator/cassandra/controller/cluster.go

```
// scaleUpRack handles scaling up for an existing Cassandra Rack.
// Calling this action implies all members of the Rack are Ready.
func (cc *ClusterController) scaleUpRack(r cassandrav1alpha1.RackSpec,
↪  c *cassandrav1alpha1.Cluster) error {

    sts, err :=
    ↪  cc.statefulSetLister.StatefulSets(c.Namespace).Get(util.StatefulSetNameForRack(
    ↪  c))
    if err != nil {
        return fmt.Errorf("error trying to scale rack %s in namespace
        ↪  %s, underlying StatefulSet not found", r.Name, c.Namespace)
    }

    logger.Infof("Attempting to scale up Rack %s", r.Name)

    err = util.ScaleStatefulSet(sts, 1, cc.kubeClient)

    if err == nil {
        cc.recorder.Event(
            c,
```

```
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackScaledUp, r.Name,
            ↪  *sts.Spec.Replicas+1),
        )
    }

    return err

}
```

This concludes the reconciliation loop for the Cluster Controller. However, we also have to implement the sidecar logic, as depicted in the sequence diagram.

As in the case of the Cluster Controller, the Sidecar is build into the rook binary as a subcommand, specifically ”rook cassandra sidecar”.

```
// scaleUpRack handles scaling up for an existing Cassandra Rack.
// Calling this action implies all members of the Rack are Ready.
func (cc *ClusterController) scaleUpRack(r cassandrav1alpha1.RackSpec, c *cassandrav1alpha

        sts, err := cc.statefulSetLister.StatefulSets(c.Namespace).Get(util.StatefulSetNam
        if err != nil {
                return fmt.Errorf("error trying to scale rack %s in namespace %s, underlyi
        }

        logger.Infof("Attempting to scale up Rack %s", r.Name)

        err = util.ScaleStatefulSet(sts, 1, cc.kubeClient)

        if err == nil {
                cc.recorder.Event(
                        c,
                        corev1.EventTypeNormal,
```

```
                        SuccessSynced,
                        fmt.Sprintf(MessageRackScaledUp, r.Name, *sts.Spec.Replicas+1),
                )
        }

        return err

}
```

The sidecar starts and gets the Pod's name and namespace from environment variables we defined earlier in the StatefulSet specification. After that, it creates a MemberController struct which contains information about the Member and clients to communicate with the Kubernetes API. It also includes a client to communicate with the Cassandra process via the Jolokia HTTP interface. This client is part of the go-nodetool [18] library that we wrote as way to programmatically issue administrative operations to Cassandra clusters, without having to use the nodetool CLI application.

pkg/operator/cassandra/sidecar/sidecar.go

```
// MemberController encapsulates all the tools the sidecar needs to
// talk to the Kubernetes API
type MemberController struct {
    // Metadata of the specific Member
    name, namespace, ip       string
    cluster, datacenter, rack string
    mode                      cassandrav1alpha1.ClusterMode

    // Clients to handle Kubernetes Objects
    kubeClient kubernetes.Interface
    rookClient rookClientset.Interface

    nodetool *nodetool.Nodetool
    queue    workqueue.RateLimitingInterface
    logger   *capnslog.PackageLogger
```

```
}

// New return a new MemberController
func New(
    name, namespace string,
    kubeClient kubernetes.Interface,
    rookClient rookClientset.Interface,
) (*MemberController, error) {

    logger := capnslog.NewPackageLogger("github.com/rook/rook",
    ↪  "sidecar")

    // Get the member's service
    var memberService *corev1.Service
    var err error
    for {
        memberService, err =
        ↪  kubeClient.CoreV1().Services(namespace).Get(name,
        ↪  metav1.GetOptions{})
        if err != nil {
            logger.Infof("Something went wrong trying to get Member
            ↪  Service %s", name)

        } else if len(memberService.Spec.ClusterIP) > 0 {
            break
        }
        // If something went wrong, wait a little and retry
        time.Sleep(500 * time.Millisecond)
    }

    // Get the Member's metadata from the Pod's labels
    pod, err := kubeClient.CoreV1().Pods(namespace).Get(name,
    ↪  metav1.GetOptions{})
```

```
    if err != nil {
        return nil, err
    }

    // Create a new nodetool interface to talk to Cassandra
    url, err := url.Parse(fmt.Sprintf("http://127.0.0.1:%d/jolokia/",
    ↪  constants.JolokiaPort))
    if err != nil {
        return nil, err
    }
    nodetool := nodetool.NewFromURL(url)

    // Get the member's cluster
    cluster, err :=
    ↪  rookClient.CassandraV1alpha1().Clusters(namespace).Get(pod.Labels[constants.Clu
    ↪  metav1.GetOptions{})
    if err != nil {
        return nil, err
    }

    m := &MemberController{
        name:       name,
        namespace:  namespace,
        ip:         memberService.Spec.ClusterIP,
        cluster:    pod.Labels[constants.ClusterNameLabel],
        datacenter: pod.Labels[constants.DatacenterNameLabel],
        rack:       pod.Labels[constants.RackNameLabel],
        mode:       cluster.Spec.Mode,
        kubeClient: kubeClient,
        rookClient: rookClient,
        nodetool:   nodetool,
        queue:
        ↪  workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter()),
```

```
        logger:     logger,
    }

    return m, nil
}
```

We notice that the structure of the Sidecar code is similar to a Controller. That is because in the future, the Sidecar will essentially function as a Controller, communicating through labels on its MemberService.

After the MemberController is created, it is started using the Run method:

pkg/operator/cassandra/sidecar/sidecar.go

```
// Run starts executing the sync loop for the sidecar
func (m *MemberController) Run(threadiness int, stopCh <-chan struct{})
↪   error {

    defer runtime.HandleCrash()

    if err := m.onStartup(); err != nil {
        return fmt.Errorf("error on startup: %s", err.Error())
    }

    <-stopCh
    m.logger.Info("Shutting down sidecar.")
    return nil

}
```

For the time being, the Sidecar only setups the configuration files on each Member startup. This is done in the `onStartup` method.

pkg/operator/cassandra/sidecar/sidecar.go

```
// onStartup is executed before the MemberController starts
// its sync loop.
```

```
func (m *MemberController) onStartup() error {

    // Setup HTTP checks
    m.logger.Info("Setting up HTTP Checks...")
    go func() {
        err := m.setupHTTPChecks()
        m.logger.Fatalf("Error with HTTP Server: %s", err.Error())
        panic("Something went wrong with the HTTP Checks")
    }()

    // Prepare config files for Cassandra
    m.logger.Infof("Generating cassandra config files...")
    if err := m.generateConfigFiles(); err != nil {
        return fmt.Errorf("error generating config files: %s",
        ↪  err.Error())
    }

    // Start the database daemon
    cmd := exec.Command(entrypointPath)
    cmd.Stderr = os.Stderr
    cmd.Stdout = os.Stdout
    cmd.Env = os.Environ()
    if err := cmd.Start(); err != nil {
        m.logger.Errorf("error starting database daemon: %s",
        ↪  err.Error())
        return err
    }

    return nil
}
```

The `onStartup` method first sets up the HTTP liveness and readiness checks that we have defined:

`onStartup`

After that, it sets up the configuration files with the correct values:

pkg/operator/cassandra/sidecar/config.go

```
// generateConfigFiles injects the default configuration files
// with our custom values.
func (m *MemberController) generateConfigFiles() error {

    var err error
    m.logger.Info("Generating config files")

    if m.mode == cassandrav1alpha1.ClusterModeScylla {
        err = m.generateScyllaConfigFiles()
    } else {
        err = m.generateCassandraConfigFiles()
    }

    return err
}
```

The sidecar checks if the Cluster is a Cassandra or Scylla Cluster and applies the appropriate configuration files. The complete code for the editing the configuration files can be found in the rook repository [3] and will not be included for brevity. Instead, we will describe the options that we changed:

cassandra.yaml: the main configuration file for Cassandra, found under ”/etc/cassandra/cassandra.yaml”.

Option	Description	Our Value
cluster_name		name from Cluster Object metadata
listen_address	The IP address or hostname that Cassandra binds to for connecting to this node.	IP of the Pod.
broadcast_address	The ”public” IP address this node uses to broadcast to other nodes outside the network or across regions.	MemberService’s ClusterIP
rpc_address	The listen address for CQL connections.	IP of the Pod.
broadcast_rpc_address	RPC address to broadcast to drivers and other Cassandra Members.	A publicly accessible IP obtained via a LoadBalancer per Member, if Cassandra needs to be accessible from outside the Cluster. Otherwise, same as broadcast_address.
endpoint_snitch	Cassandra uses the snitch to discover the Cluster topology. The snitch will read the Member’s topology information from a local file and then gossip that information around the Cluster so other Members find out.	GossipingPropertyFileSnitch. Also need to pass DC, Rack values to Pod.
seed_provider	The addresses of hosts designated as contact points in the cluster. A new instance contacts one of the Members in the -seeds list to join the Cassandra Cluster.	The sidecar gets MemberServices with ”cassandra.rook.io/seed”, ”cassandra.rook.io/cluster” labels set accordingly and provides their ClusterIP through the –seeds list.

cassandra-env.sh:

- Load Jolokia for JMX <–> HTTP communication:

```
JVM_OPTS="$JVM_OPTS
↪ -javaagent:/mnt/shared/plugins/jolokia.jar=port=<agent_port>,host=<node_ip>"
```

- Specify the correct amount of RAM for Cassandra to use, by setting the MAX_-HEAPSIZE and HEAP_NEWSIZE environment variables, according to the recommendations in Cassandra [3] [4]

cassandra-rackdc.properties: File containing the Member's topology, which includes Rack and Datacenter name. Used by the GossipingPropertyFileSnitch.

Option	Our Value
dc	Datacenter name, sidecar discovers it on startup.
rack	Rack name, sidecar discovers it on startup.
prefer_local	This option tells Cassandra to prefer talking through listen_address rather than broadcast_address, for Members in the same Datacenter. Set to false, as the ephemeral nature of IPs in Kubernetes can cause problems.

The configuration files for Scylla are very similar. After editing the configuration files, the sidecar starts Cassandra in a new process:

pkg/operator/cassandra/sidecar/sidecar.go

```
// onStartup is executed before the MemberController starts
// its sync loop.
```

[3] MAX_HEAPSIZE https://github.com/apache/cassandra/blob/dccf53061a61e7c632669c60cd94626e405518e9/conf/cassandra-env.sh#L73

[4] HEAP_NEWSIZE https://github.com/apache/cassandra/blob/dccf53061a61e7c632669c60cd94626e405518e9/conf/cassandra-env.sh#L81-L86

```
func (m *MemberController) onStartup() error {

    ...

    // Start the database daemon
    cmd := exec.Command(entrypointPath)
    cmd.Stderr = os.Stderr
    cmd.Stdout = os.Stdout
    cmd.Env = os.Environ()
    if err := cmd.Start(); err != nil {
        m.logger.Errorf("error starting database daemon: %s",
        ↪  err.Error())
        return err
    }

    return nil
}
```

This concludes the Cluster Creation & Scale Up workflow.

4.6.2 Cluster Scale Down

First of all, let's remember the design we created in subsection 3.9.2. The sequence diagram of the Cluster Creation and Scale Up workflow is the following:

Following the sequence diagram, we first need to detect when a Cluster needs to scale down in the Cluster Controller. We add the following code to the `syncCluster` function:

pkg/operator/cassandra/controller/cluster.go

```
// SyncCluster checks the Status and performs reconciliation for
// the given Cassandra Cluster.
func (cc *ClusterController) syncCluster(c *cassandrav1alpha1.Cluster)
↪  error {
```

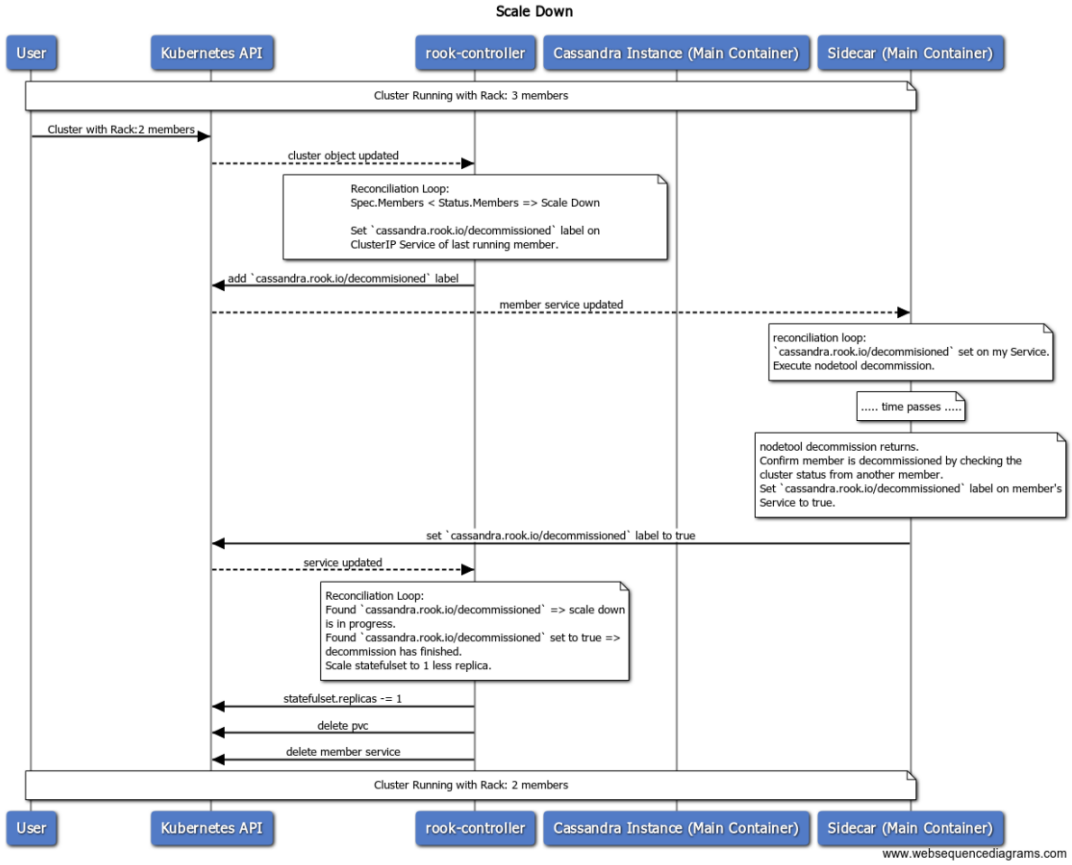


Figure 4.3: *Cluster Scale Down Sequence Diagram, as seen in subsection 3.9.2*

```
// Check if any rack isn't created
for _, rack := range c.Spec.Datacenter.Racks {
    // For each rack, check if a status entry exists
    if _, ok := c.Status.Racks[rack.Name]; !ok {
        logger.Infof("Attempting to create Rack %s", rack.Name)
        err := cc.createRack(rack, c)
        return err
    }
}

// Check if there is a scale-down in progress
for _, rack := range c.Spec.Datacenter.Racks {
    if util.IsRackConditionTrue(c.Status.Racks[rack.Name],
    ↪  cassandrav1alpha1.RackConditionTypeMemberLeaving) {
```

```
            // Resume scale down
            err := cc.scaleDownRack(rack, c)
            return err
        }
    }

    // Check that all racks are ready before taking any action
    for _, rack := range c.Spec.Datacenter.Racks {
        rackStatus := c.Status.Racks[rack.Name]
        if rackStatus.Members != rackStatus.ReadyMembers {
            logger.Infof("Rack %s is not ready, %+v", rack.Name,
            ↪  *rackStatus)
            return nil
        }
    }

    // Check if any rack needs to scale down
    for _, rack := range c.Spec.Datacenter.Racks {
        if rack.Members < c.Status.Racks[rack.Name].Members {
            // scale down
            err := cc.scaleDownRack(rack, c)
            return err
        }
    }

    // Check if any rack needs to scale up
    for _, rack := range c.Spec.Datacenter.Racks {

        if rack.Members > c.Status.Racks[rack.Name].Members {
            logger.Infof("Attempting to scale rack %s", rack.Name)
            err := cc.scaleUpRack(rack, c)
            return err
        }
```

```
    }

    return nil
}
```

There are 2 possibilities:

1. The Rack has more Members than the Spec, so it needs to scale down. We detect this by comparing the Cluster Spec with the Status.

2. The Rack is scaling down at the moment. We detect this with the ”MemberLeaving” Condition in the Cluster Status. Conditions represent the latest available observations of an Object’s state. We have added the following code to our Cluster struct definition to support them:

```
// SyncCluster checks the Status and performs reconciliation for
// the given Cassandra Cluster.
func (cc *ClusterController) syncCluster(c *cassandrav1alpha1.Cluster) error {

        // Check if any rack isn't created
        for _, rack := range c.Spec.Datacenter.Racks {
                // For each rack, check if a status entry exists
                if _, ok := c.Status.Racks[rack.Name]; !ok {
                        logger.Infof("Attempting to create Rack %s", rack.Name)
                        err := cc.createRack(rack, c)
                        return err
                }
        }

        // Check if there is a scale-down in progress
        for _, rack := range c.Spec.Datacenter.Racks {
                if util.IsRackConditionTrue(c.Status.Racks[rack.Name], cassandrav1alp
                        // Resume scale down
                        err := cc.scaleDownRack(rack, c)
```

```
                return err
        }
}

// Check that all racks are ready before taking any action
for _, rack := range c.Spec.Datacenter.Racks {
        rackStatus := c.Status.Racks[rack.Name]
        if rackStatus.Members != rackStatus.ReadyMembers {
                logger.Infof("Rack %s is not ready, %+v", rack.Name, *rac
                return nil
        }
}

// Check if any rack needs to scale down
for _, rack := range c.Spec.Datacenter.Racks {
        if rack.Members < c.Status.Racks[rack.Name].Members {
                // scale down
                err := cc.scaleDownRack(rack, c)
                return err
        }
}

// Check if any rack needs to scale up
for _, rack := range c.Spec.Datacenter.Racks {

        if rack.Members > c.Status.Racks[rack.Name].Members {
                logger.Infof("Attempting to scale rack %s", rack.Name)
                err := cc.scaleUpRack(rack, c)
                return err
        }
}

return nil
```

```
    }
```

The ”MemberLeaving” condition is set by the updateStatus function:

pkg/operator/cassandra/controller/cluster.go

```
// UpdateStatus updates the status of the given Cassandra Cluster.
// It doesn't post the result to the API Server yet.
// That will be done at the end of the sync loop.
func (cc *ClusterController) updateStatus(c *cassandrav1alpha1.Cluster)
↪  error {

    clusterStatus := cassandrav1alpha1.ClusterStatus{
        Racks: map[string]*cassandrav1alpha1.RackStatus{},
    }
    logger.Infof("Updating Status for cluster %s in namespace %s",
    ↪  c.Name, c.Namespace)

    for _, rack := range c.Spec.Datacenter.Racks {

        status := &cassandrav1alpha1.RackStatus{}

        // Get corresponding StatefulSet from lister
        sts, err := cc.statefulSetLister.StatefulSets(c.Namespace).
            Get(util.StatefulSetNameForRack(rack, c))

        ...

        // Update Scaling Down condition
        services, err := util.GerMemberServicesForRack(rack, c,
        ↪  cc.serviceLister)
        if err != nil {
            return fmt.Errorf("error trying to get Pods for rack %s",
            ↪  rack.Name)
        }
```

```
        for _, svc := range services {
            // Check if there is a decommission in progress
            if _, ok := svc.Labels[constants.DecommissionLabel]; ok {
                // Add MemberLeaving Condition to rack status
                status.Conditions = append(status.Conditions,
                ↪  cassandrav1alpha1.RackCondition{
                    Type:
                    ↪  cassandrav1alpha1.RackConditionTypeMemberLeaving,
                    Status: cassandrav1alpha1.ConditionTrue,
                })
                // Sanity check. Only the last member should be
                ↪  decommissioning.
                index, err := util.IndexFromName(svc.Name)
                if err != nil {
                    return err
                }
                if index != status.Members-1 {
                    return fmt.Errorf("only last member of each rack
                    ↪  should be decommissioning, but %d-th member of
                    ↪  %s found decommissioning while rack had %d
                    ↪  members", index, rack.Name, status.Members)
                }
            }
        }

        // Update Status for Rack
        clusterStatus.Racks[rack.Name] = status
    }

    c.Status = clusterStatus
    return nil
}
```

When either case of Scale Down is detected, the `scaleDownRack` function is called:

pkg/operator/cassandra/controller/cluster.go

```
// scaleDownRack handles scaling down for an existing Cassandra Rack.
// Calling this action implies all members of the Rack are Ready.
func (cc *ClusterController) scaleDownRack(r
↪  cassandrav1alpha1.RackSpec, c *cassandrav1alpha1.Cluster) error {

    logger.Infof("Scaling down rack %s", r.Name)

    // Get the current actual number of Members
    members := c.Status.Racks[r.Name].Members

    // Find the member to decommission
    memberName := fmt.Sprintf("%s-%d", util.StatefulSetNameForRack(r,
    ↪  c), members-1)
    logger.Infof("Member of interest: %s", memberName)
    memberService, err :=
    ↪  cc.serviceLister.Services(c.Namespace).Get(memberName)
    if err != nil {
        return fmt.Errorf("error trying to get Member Service %s: %s",
        ↪  memberName, err.Error())
    }

    // Check if there was a scale down in progress that has completed.
    if memberService.Labels[constants.DecommissionLabel] ==
    ↪  constants.LabelValueTrue {

        logger.Infof("Found decommissioned member: %s", memberName)

        // Get rack's statefulset
        stsName := util.StatefulSetNameForRack(r, c)
        sts, err :=
        ↪  cc.statefulSetLister.StatefulSets(c.Namespace).Get(stsName)
```

```
        if err != nil {
            return fmt.Errorf("error trying to get StatefulSet %s",
            ↪  stsName)
        }
        // Scale the statefulset
        err = util.ScaleStatefulSet(sts, -1, cc.kubeClient)
        if err != nil {
            return fmt.Errorf("error trying to scale down StatefulSet
            ↪  %s", stsName)
        }
        // Cleanup is done on each sync loop, no need to do anything
        ↪  else here

        cc.recorder.Event(
            c,
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackScaledDown, r.Name, members-1),
        )
        return nil
    }

    logger.Infof("Checking for scale down. Desired: %d. Actual: %d",
    ↪  r.Members, c.Status.Racks[r.Name].Members)
    // Then, check if there is a requested scale down.
    if r.Members < c.Status.Racks[r.Name].Members {

        logger.Infof("Scale down requested, member %s will
        ↪  decommission", memberName)
        // Record the intent to decommission the member
        old := memberService.DeepCopy()
        memberService.Labels[constants.DecommissionLabel] =
        ↪  constants.LabelValueFalse
```

```
        if err := util.PatchService(old, memberService, cc.kubeClient);
        ↪  err != nil {
            return fmt.Errorf("error patching member service %s: %s",
            ↪  memberName, err.Error())
        }

        cc.recorder.Event(
            c,
            corev1.EventTypeNormal,
            SuccessSynced,
            fmt.Sprintf(MessageRackScaleDownInProgress, r.Name,
            ↪  members-1),
        )
    }

    return nil
}
```

The `scaleDownRack` function detects if the Rack is scaling down or needs to initiate a scale down.

- If it is scaling down and has finished, it removes a replica from the StatefulSet.
- If it is scaling down and hasn't finished, it does nothing.
- If it needs to initiate a scale down, it write the "cassandra.rook.io/decommissioned=false" label on the MemberService of the Member with the highest index, which is the one that will get deleted when the StatefulSet scales down.

Finally, for cleaning up the Persistent Volume Claim and Member Service of the Member that was deleted, we check for remaining Objects on the start of each Controller sync:

pkg/operator/cassandra/controller/sync.go

```
// Sync attempts to sync the given Cassandra Cluster.
// NOTE: the Cluster Object is a DeepCopy. Modify at will.
```

```
func (cc *ClusterController) Sync(c *cassandrav1alpha1.Cluster) error {

    // Before syncing, ensure that all StatefulSets are up-to-date
    stale, err := util.StatefulSetStatusesStale(c,
    ↪  cc.statefulSetLister)
    if err != nil {
        return err
    }
    if stale {
        return nil
    }

    // Cleanup Cluster resources
    if err := cc.cleanup(c); err != nil {
        cc.recorder.Event(
            c,
            corev1.EventTypeWarning,
            ErrSyncFailed,
            MessageCleanupFailed,
        )
    }

    ...

}
```

pkg/operator/cassandra/controller/cleanup.go

```
// cleanup deletes all resources remaining because of cluster scale
↪  downs
func (cc *ClusterController) cleanup(c *cassandrav1alpha1.Cluster)
↪  error {

    for _, r := range c.Spec.Datacenter.Racks {
```

```
services, err :=
↪  cc.serviceLister.Services(c.Namespace).List(util.RackSelector(r,
↪  c))
if err != nil {
    return fmt.Errorf("error listing member services: %s",
    ↪  err.Error())
}
// Get rack status. If it doesn't exist, the rack isn't yet
↪  created.
stsName := util.StatefulSetNameForRack(r, c)
sts, err :=
↪  cc.statefulSetLister.StatefulSets(c.Namespace).Get(stsName)
if apierrors.IsNotFound(err) {
    continue
}
if err != nil {
    return fmt.Errorf("error getting statefulset %s: %s",
    ↪  stsName, err.Error())
}
memberCount := *sts.Spec.Replicas
memberServiceCount := int32(len(services))
// If there are more services than members, some services need
↪  to be cleaned up
if memberServiceCount > memberCount {
    maxIndex := memberCount - 1
    for _, svc := range services {
        svcIndex, err := util.IndexFromName(svc.Name)
        if err != nil {
            logger.Errorf("Unexpected error while parsing index
            ↪  from name %s : %s", svc.Name, err.Error())
            continue
        }
        if svcIndex > maxIndex {
```

```
                    err := cc.cleanupMemberResources(svc.Name, r, c)
                    if err != nil {
                        return fmt.Errorf("error cleaning up member
                        ↪  resources: %s", err.Error())
                    }
                }
            }
        }
    }
    logger.Infof("%s/%s - Successfully cleaned up cluster.",
    ↪  c.Namespace, c.Name)
    return nil
}

// cleanupMemberResources deletes all resources associated with a given
↪  member.
// Currently those are :
//  - A PVC
//  - A ClusterIP Service
func (cc *ClusterController) cleanupMemberResources(memberName string,
↪  r cassandrav1alpha1.RackSpec, c *cassandrav1alpha1.Cluster) error {

    logger.Infof("%s/%s - Cleaning up resources for member %s",
    ↪  c.Namespace, c.Name, memberName)
    // Delete PVC
    if len(r.Storage.VolumeClaimTemplates) > 0 {
        // PVC naming convention for StatefulSets is
        ↪  <volumeClaimTemplate.Name>-<pod.Name>
        pvcName := fmt.Sprintf("%s-%s",
        ↪  r.Storage.VolumeClaimTemplates[0].Name, memberName)
        err :=
        ↪  cc.kubeClient.CoreV1().PersistentVolumeClaims(c.Namespace).Delete(pvcName,
        ↪  &metav1.DeleteOptions{})
```

```
        if err != nil && !apierrors.IsNotFound(err) {
            return fmt.Errorf("error deleting pvc %s: %s", pvcName,
            ↪  err.Error())
        }
    }

    // Delete Member Service
    err :=
    ↪  cc.kubeClient.CoreV1().Services(c.Namespace).Delete(memberName,
    ↪  &metav1.DeleteOptions{})
    if err != nil {
        return fmt.Errorf("error deleting member service %s: %s",
        ↪  memberName, err.Error())
    }
    return nil
}
```

4.6.2.1 Sidecar

We have completed the necessary changes in the Cluster Controller and now we need to edit the Sidecar code. The Sidecar should:

1. Watch the Member's Service.
2. If the "cassandra.rook.io/decommissioned=false" label is set, decommission the Member.

To watch the Member's Service, we use an Informer, as in the Cluster Controller. In this case, we use a FilteredInformer to only watch the Member that is of interest to us:

cmd/rook/cassandra/sidecar.go

```
func startSidecar(cmd *cobra.Command, args []string) error {
```

```
...

// kubeInformerFactory watches resources with:
    // namespace: podNamespace
    // name: podName
    kubeInformerFactory :=
    ↪  kubeinformers.NewSharedInformerFactoryWithOptions(
        kubeClient,
        resyncPeriod,
        kubeinformers.WithNamespace(podNamespace),
        kubeinformers.WithTweakListOptions(tweakListOptionsFunc),
    )

    mc, err := sidecar.New(
        podName,
        podNamespace,
        kubeClient,
        rookClient,
        kubeInformerFactory.Core().V1().Services(),
    )

...

}
```

The Sidecar Controller scaffolding is very similar to the one for Cluster Controller:

pkg/operator/cassandra/sidecar/sidecar.go

```
// New return a new MemberController
func New(
    name, namespace string,
    kubeClient kubernetes.Interface,
    rookClient rookClientset.Interface,
```

```
    serviceInformer coreinformers.ServiceInformer,
) (*MemberController, error) {

    logger := capnslog.NewPackageLogger("github.com/rook/rook",
    ↪ "sidecar")

    // Get the member's service
    var memberService *corev1.Service
    var err error
    for {
        memberService, err =
        ↪ kubeClient.CoreV1().Services(namespace).Get(name,
        ↪ metav1.GetOptions{})
        if err != nil {
            logger.Infof("Something went wrong trying to get Member
            ↪ Service %s", name)

        } else if len(memberService.Spec.ClusterIP) > 0 {
            break
        }
        // If something went wrong, wait a little and retry
        time.Sleep(500 * time.Millisecond)
    }

    // Get the Member's metadata from the Pod's labels
    pod, err := kubeClient.CoreV1().Pods(namespace).Get(name,
    ↪ metav1.GetOptions{})
    if err != nil {
        return nil, err
    }

    // Create a new nodetool interface to talk to Cassandra
```

```
    url, err := url.Parse(fmt.Sprintf("http://127.0.0.1:%d/jolokia/",
    ↪  constants.JolokiaPort))
    if err != nil {
        return nil, err
    }
    nodetool := nodetool.NewFromURL(url)

    // Get the member's cluster
    cluster, err :=
    ↪  rookClient.CassandraV1alpha1().Clusters(namespace).Get(pod.Labels[constants.Clu
    ↪  metav1.GetOptions{})
    if err != nil {
        return nil, err
    }

    m := &MemberController{
        name:                name,
        namespace:           namespace,
        ip:                  memberService.Spec.ClusterIP,
        cluster:             pod.Labels[constants.ClusterNameLabel],
        datacenter:          pod.Labels[constants.DatacenterNameLabel],
        rack:                pod.Labels[constants.RackNameLabel],
        mode:                cluster.Spec.Mode,
        kubeClient:          kubeClient,
        rookClient:          rookClient,
        serviceLister:       serviceInformer.Lister(),
        serviceListerSynced: serviceInformer.Informer().HasSynced,
        nodetool:            nodetool,
        queue:
        ↪  workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter()),
        logger:              logger,
    }
```

```
serviceInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        svc := obj.(*corev1.Service)
        if svc.Name != m.name {
            logger.Errorf("Lister returned unexpected service %s",
            ↪ svc.Name)
            return
        }
        m.enqueueMemberService(svc)
    },
    UpdateFunc: func(old, new interface{}) {
        oldService := old.(*corev1.Service)
        newService := new.(*corev1.Service)
        if oldService.ResourceVersion == newService.ResourceVersion
        ↪ {
            return
        }
        if reflect.DeepEqual(oldService.Labels, newService.Labels)
        ↪ {
            return
        }
        logger.Infof("New event for my MemberService %s",
        ↪ newService.Name)
        m.enqueueMemberService(newService)
    },
    DeleteFunc: func(obj interface{}) {
        svc := obj.(*corev1.Service)
        if svc.Name == m.name {
            logger.Errorf("Unexpected deletion of MemberService
            ↪ %s", svc.Name)
        }
    },
})
```

```
    return m, nil
}

// Run starts executing the sync loop for the sidecar
func (m *MemberController) Run(threadiness int, stopCh <-chan struct{})
↪   error {

    defer runtime.HandleCrash()

    if ok := cache.WaitForCacheSync(stopCh, m.serviceListerSynced); !ok
    ↪   {
        return fmt.Errorf("failed to wait for caches to sync")
    }

    if err := m.onStartup(); err != nil {
        return fmt.Errorf("error on startup: %s", err.Error())
    }

    m.logger.Infof("Main event loop")
    go wait.Until(m.runWorker, time.Second, stopCh)

    <-stopCh
    m.logger.Info("Shutting down sidecar.")
    return nil

}

func (m *MemberController) runWorker() {
    for m.processNextWorkItem() {
    }
}
```

```
func (m *MemberController) processNextWorkItem() bool {
    obj, shutdown := m.queue.Get()

    if shutdown {
        return false
    }

    err := func(obj interface{}) error {
        defer m.queue.Done(obj)
        key, ok := obj.(string)
        if !ok {
            m.queue.Forget(obj)
            runtime.HandleError(fmt.Errorf("expected string in queue
            ↪  but got %#v", obj))
        }
        if err := m.syncHandler(key); err != nil {
            m.queue.AddRateLimited(key)
            return fmt.Errorf("error syncing '%s', requeueing: %s",
            ↪  key, err.Error())
        }
        m.queue.Forget(obj)
        m.logger.Infof("Successfully synced '%s'", key)
        return nil
    }(obj)

    if err != nil {
        runtime.HandleError(err)
        return true
    }

    return true
}
```

```
func (m *MemberController) syncHandler(key string) error {
    // Convert the namespace/name string into a distinct namespace and
    ↪ name.
    namespace, name, err := cache.SplitMetaNamespaceKey(key)
    if err != nil {
        runtime.HandleError(fmt.Errorf("invalid resource key: %s",
        ↪ key))
        return nil
    }

    // Get the Cluster resource with this namespace/name
    svc, err := m.serviceLister.Services(namespace).Get(name)
    if err != nil {
        // The Cluster resource may no longer exist, in which case we
        ↪ stop processing.
        if apierrors.IsNotFound(err) {
            runtime.HandleError(fmt.Errorf("member service '%s' in work
            ↪ queue no longer exists", key))
            return nil
        }
        return fmt.Errorf("unexpected error while getting member
        ↪ service object: %s", err)
    }

    m.logger.Infof("handling member service object: %+v",
    ↪ spew.Sdump(svc))
    err = m.Sync(svc)

    return err
}
```

Finally, the reconciliation loop for the Sidecar is very simple:

pkg/operator/cassandra/sidecar/sync.go

```
func (m *MemberController) Sync(memberService *v1.Service) error {

    // Check if member must decommission
    if decommission, ok :=
    ↪  memberService.Labels[constants.DecommissionLabel]; ok {
        // Check if member has already decommissioned
        if decommission == constants.LabelValueTrue {
            return nil
        }
        // Else, decommission member
        if err := m.nodetool.Decommission(); err != nil {
            m.logger.Errorf("Error during decommission: %s",
            ↪  err.Error())
        }
        // Confirm memberService has been decommissioned
        if opMode, err := m.nodetool.OperationMode(); err != nil ||
        ↪  opMode != nodetool.NodeOperationModeDecommissioned {
            return fmt.Errorf("error during decommission, operation
            ↪  mode: %s, error: %v", opMode, err)
        }
        // Update Label
        old := memberService.DeepCopy()
        memberService.Labels[constants.DecommissionLabel] =
        ↪  constants.LabelValueTrue
        if err := util.PatchService(old, memberService, m.kubeClient);
        ↪  err != nil {
            return fmt.Errorf("error patching MemberService, %s",
            ↪  err.Error())
        }

    }
```

```
    return nil
}
```

This concludes the implementation of the Cluster Scale Down.

4.6.3 Local Disks

For the most part of its history, Kubernetes has focused on using Network Attached Storage solutions that provide replication and fault-tolerance of the Persistent Volumes. This fault-tolerance however, comes with a reduction in performance.

On the other hand, we have the solution of using fast, local disks with much better performance and cost. These local disks are ephemeral and not replicated.

Cassandra handles replication in the application level. Because of this, it is unnecessary to use slower and more expensive Network Attached Storage. Instead, we will try to use Local Storage with Kubernetes.

5 Evaluation

In this chapter, we evaluate the functionality of our Cassandra Operator by running it in a GKE Kubernetes Cluster and evaluating its response in the scenarios of Cluster Deployment, Scale Up, Scale Down and Loss of a Local Disk.

5.1 Tools, Methodology and Environment

All of the experiments are conducted in a Kubernetes environment, provided by the Google Kubernetes Engine service. To interact with the Kubernetes cluster, we use *kubectl*, a command line interface for running commands against Kubernetes clusters. It can perform various actions like getting and editing Kubernetes Objects, accessing logs from Pods and providing shell access inside Pods. We also make use of local disks that provide superior performance but are confined to the Node they live on.

The command used to setup the environment is:

gke-setup-script.sh

```
#!/bin/bash

set -e

GCP_USER=${1:-"yanniszarkadas@gmail.com"}
GCP_PROJECT=${2:-"gke-demo-226716"}
```

```
gcloud beta container clusters create "cluster" \
--project "$GCP_PROJECT" \
--region "europe-west1" \
--node-locations "europe-west1-b,europe-west1-c" \
--cluster-version "1.11.5-gke.5" \
--machine-type "n1-standard-4" \
--image-type "UBUNTU" \
--disk-type "pd-ssd" --disk-size "20" \
--local-ssd-count "1" \
--num-nodes "4" \
--no-enable-autoupgrade --no-enable-autorepair


# gcloud: Get credentials for new cluster
echo "Getting credentials for newly created cluster..."
gcloud container clusters get-credentials cluster --region=europe-west1

# Setup GKE RBAC
echo "Setting up GKE RBAC..."
kubectl create clusterrolebinding cluster-admin-binding --clusterrole
↪  cluster-admin --user "$GCP_USER"
```

In addition, we use Helm to install the Local Volume Static Provisioner that will expose the local SSDs as Kubernetes Persistent Volumes. The Storage Class that contains those volumes is named ”local-disks”.

gke-setup-script.sh

```
echo "Checking if helm is present on the machine..."
if ! hash helm 2>/dev/null; then
  echo "You need to install helm. See:
  ↪  https://docs.helm.sh/using_helm/#installing-helm"
  exit 1
```

```
fi

# Setup Tiller
echo "Setting up Tiller..."
helm init
kubectl create serviceaccount --namespace kube-system tiller
kubectl create clusterrolebinding tiller-cluster-rule
↪  --clusterrole=cluster-admin --serviceaccount=kube-system:tiller
kubectl patch deploy --namespace kube-system tiller-deploy -p
↪  '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'

# Wait for Tiller to become ready
until [[ $(kubectl get deployment tiller-deploy -n kube-system -o
↪  'jsonpath={.status.readyReplicas}') -eq 1 ]];
do
    echo "Waiting for Tiller pod to become Ready..."
    sleep 5
done

# Install local volume provisioner
echo "Installing local volume provisioner..."
helm install --name local-provisioner provisioner
echo "Your disks are ready to use."
```

For the Cassandra Operator, we use the following manifest for a Cluster of 3 Members in europe-west1-b:

cassandra-custom-resource.yaml

```
# Cassandra Cluster
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
```

```
  namespace: rook-cassandra
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: europe-west1
    racks:
      - name: europe-west1-b
        members: 3
        storage:
          volumeClaimTemplates:
                - metadata:
                    name: rook-cassandra-data
                  spec:
                    storageClassName: local-disks
                    resources:
                      requests:
                        storage: 350Gi
        resources:
          requests:
            cpu: 2
            memory: 8Gi
          limits:
            cpu: 2
            memory: 8Gi
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: failure-domain.beta.kubernetes.io/zone
                    operator: In
                    values:
```

```
                - europe-west1-b
```

Along with the Custom Resource, we submit a Role, RoleBinding and ServiceAccount for the Cluster's Members. Those Objects are needed in order to give Members the necessary RBAC permissions to function correctly.

We also run the Cassandra Operator in a StatefulSet:

cassandra-operator.yaml

```yaml
# Namespace where Cassandra Operator will live
apiVersion: v1
kind: Namespace
metadata:
  name: rook-cassandra-system

---

# Cassandra Cluster CRD
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: clusters.cassandra.rook.io
spec:
  group: cassandra.rook.io
  names:
    kind: Cluster
    listKind: ClusterList
    plural: clusters
    singular: cluster
  scope: Namespaced
  version: v1alpha1
  # openapi validation omitted for brevity

---
```

```
# ClusterRole for cassandra-operator.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: rook-cassandra-operator
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - list
      - watch
      - delete
  - apiGroups:
      - ""
    resources:
      - services
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - persistentvolumeclaims
    verbs:
      - get
      - create
      - delete
  - apiGroups:
      - ""
    resources:
```

```
      - nodes
      - persistentvolumes
    verbs:
      - get
  - apiGroups:
      - apps
    resources:
      - statefulsets
    verbs:
      - "*"
  - apiGroups:
      - policy
    resources:
      - poddisruptionbudgets
    verbs:
      - create
  - apiGroups:
      - cassandra.rook.io
    resources:
      - "*"
    verbs:
      - "*"
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - create
      - update
      - patch
---
# ServiceAccount for cassandra-operator. Serves as its authorization
↪  identity.
```

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# Bind cassandra-operator ServiceAccount with ClusterRole.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rook-cassandra-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: rook-cassandra-operator
subjects:
- kind: ServiceAccount
  name: rook-cassandra-operator
  namespace: rook-cassandra-system
---
# cassandra-operator StatefulSet.
 apiVersion: apps/v1
 kind: StatefulSet
 metadata:
   name: rook-cassandra-operator
   namespace: rook-cassandra-system
   labels:
     app: rook-cassandra-operator
 spec:
   replicas: 1
   serviceName: "non-existent-service"
   selector:
     matchLabels:
```

```
      app: rook-cassandra-operator
  template:
    metadata:
      labels:
        app: rook-cassandra-operator
    spec:
      serviceAccountName: rook-cassandra-operator
      containers:
      - name: rook-cassandra-operator
        image: yanniszark/rook-cassandra-operator:meetup-presentation
        imagePullPolicy: "Always"
        args: ["cassandra", "operator"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

5.2 Results

5.2.1 Creation and Scale-Up of a Casssandra Cluster

In the beginning, we create a Cassandra Cluster with 3 Members in Availability Zone europe-west1-b:

cluster.yaml

```
# Cassandra Cluster
apiVersion: cassandra.rook.io/v1alpha1
```

```
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: europe-west1
    racks:
      - name: europe-west1-b
        members: 3
        storage:
          volumeClaimTemplates:
                - metadata:
                    name: rook-cassandra-data
                  spec:
                    storageClassName: local-disks
                    resources:
                      requests:
                        storage: 350Gi
        resources:
          requests:
            cpu: 2
            memory: 8Gi
          limits:
            cpu: 2
            memory: 8Gi
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
```

```
            - key: failure-domain.beta.kubernetes.io/zone
              operator: In
              values:
                - europe-west1-b
```

The Cassandra Operator is notified that a new Cassandra Cluster Object exists and starts taking actions to ensure that the Cluster is created and scaled up.

With kubectl, we watch for the Objects created by our Operator. First, we see the StatefulSet created for Rack europe-west1-b:

```
yannis@dev-pc:~$ kubectl get statefulsets
NAME                                          DESIRED   CURRENT   AGE
rook-cassandra-europe-west1-europe-west1-b    3         3         1h
```

The Pods created by the StatefulSet:

```
yannis@dev-pc:~$ kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
rook-cassandra-europe-west1-europe-west1-b-0  1/1     Running   0          1h
rook-cassandra-europe-west1-europe-west1-b-1  1/1     Running   0          1h
rook-cassandra-europe-west1-europe-west1-b-2  1/1     Running   0          1h
```

The ClusterIP Services created for each Member as well as the client service:

```
yannis@dev-pc:~$ kubectl get service
NAME                                          TYPE        CLUSTER-IP      EXTERNAL-IP
rook-cassandra-client                         ClusterIP   None            <none>
rook-cassandra-europe-west1-europe-west1-b-0  ClusterIP   10.31.255.200   <none>
rook-cassandra-europe-west1-europe-west1-b-1  ClusterIP   10.31.241.133   <none>
rook-cassandra-europe-west1-europe-west1-b-2  ClusterIP   10.31.243.96    <none>
```

The Status and Events for the Cassandra Cluster are updated by our operator:

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
```

```
  Racks:
    Europe - West 1 - B:
      Members:        3
      Ready Members:  3
Events:               <none>
```

To test scaling up, we add a new rack in availability zone europe-west1-c, with the command `kubectl edit clusters.cassandra.rook.io rook-cassandra`.

cluster.yaml

```yaml
# Cassandra Cluster
apiVersion: cassandra.rook.io/v1alpha1
kind: Cluster
metadata:
  name: rook-cassandra
  namespace: rook-cassandra
spec:
  version: 3.11.1
  mode: cassandra
  datacenter:
    name: europe-west1
    racks:
      - name: europe-west1-b
        members: 3
        storage:
          volumeClaimTemplates:
                - metadata:
                    name: rook-cassandra-data
                  spec:
                    storageClassName: local-disks
                    resources:
                      requests:
                        storage: 350Gi
        resources:
```

```
      requests:
        cpu: 2
        memory: 8Gi
      limits:
        cpu: 2
        memory: 8Gi
    placement:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                  - europe-west1-b
  - name: europe-west1-c
    members: 2
    storage:
      volumeClaimTemplates:
            - metadata:
                name: rook-cassandra-data
              spec:
                storageClassName: local-disks
                resources:
                  requests:
                    storage: 350Gi
    resources:
      requests:
        cpu: 2
        memory: 8Gi
      limits:
        cpu: 2
        memory: 8Gi
```

```
        placement:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: failure-domain.beta.kubernetes.io/zone
                    operator: In
                    values:
                      - europe-west1-c
```

From the Status and Events of the Cluster Object, we observe the progress of the scale up action:

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
  Racks:
    Europe - West 1 - B:
      Members:        3
      Ready Members:  3
    Europe - West 1 - C:
      Members:        2
      Ready Members:  2
Events:
  Type    Reason  Age    From                  Message
  ----    ------  ----   ----                  -------
  Normal  Synced  5m15s  cassandra-controller  Rack europe-west1-c created
  Normal  Synced  5m15s  cassandra-controller  Rack europe-west1-c scaled up to 1 members
  Normal  Synced  4m36s  cassandra-controller  Rack europe-west1-c scaled up to 2 members
```

With this, we have created a Cassandra Cluster spanning multiple Availability Zones, which was impossible using a StatefulSet.

5.2.2 Scale-Down of a Cassandra Cluster

After scaling our cluster to 5 Members, we will scale it down to 3 Members again. More specifically, we will reduce the Rack europe-west1-b to 1 Member. Once again, we will use the command `kubectl edit clusters.cassandra.rook.io rook-cassandra`.

After changing the number of Members, we observe that the Operator once again refreshes the Status of the Cassandra Cluster Object and informs us of the situation at the moment.

```
yannis@dev-pc:~$ kubectl describe clusters.cassandra.rook.io rook-cassandra
Status:
  Racks:
    Europe - West 1 - B:
      Members:        1
      Ready Members:  1
    Europe - West 1 - C:
      Members:        2
      Ready Members:  2
Events:
  Type    Reason  Age                    From                  Message
  ----    ------  ----                   ----                  -------
  Normal  Synced  16m                    cassandra-controller  Rack europe-west1-c crea
  Normal  Synced  16m                    cassandra-controller  Rack europe-west1-c scal
  Normal  Synced  15m                    cassandra-controller  Rack europe-west1-c scal
  Normal  Synced  9m32s (x3 over 9m53s)  cassandra-controller  Rack europe-west1-b scal
  Normal  Synced  8m52s                  cassandra-controller  Rack europe-west1-b scal
  Normal  Synced  8m29s (x8 over 8m52s)  cassandra-controller  Rack europe-west1-b scal
  Normal  Synced  7m51s                  cassandra-controller  Rack europe-west1-b scal
```

After a while, we observe that the Cluster has scaled down. By running ”nodetool status” in one of the Members, we confirm that 2 Members have left the Cassandra Cluster.

In conclusion, we implemented the scale-down sequence correctly, as it would be done by a human administrator.

5.2.3 Loss of a Local Disk

We remind that this particular Cassandra Cluster is using local disks. Following from our design for coping with Node failures (which have local disks on them), we will simulate the failure of a Kubernetes Node. To do so, we find a Node that a Cassandra Member runs on and delete it from the Kubernetes API.

After deleting the Node and waiting a while, we will see informative Events in the Cluster Object. In a while, a new Member will be up and running, which will replace the fallen Member and stream its data from other replicas. Finally, we can look at the new Member's logs to confirm that the Member replaced the old one. In addition, the "nodetool status" command will give us information and tell us if the replacement was successful.

6 Conclusion

In this final chapter, we make an overall assessment of our implementation and outline a few directions for further improvement that we deem worthy of investigation.

6.1 Concluding Remarks

All in all, we have managed to meet the expectations of creating an open-source vendor-neutral management layer for Cassandra. While there is still work to do, mainly for Day-2 operations (backups, restores), many core features were designed and implemented.

Kubernetes Operators were proven to be a very flexible way of extending Kubernetes to correctly manage a Stateful application. In this author's opinion, they are not equal to a managed offering and still require some attention from people with knowledge of the app that's running. However, they are still extremely useful at automating a lot of the repetitive stuff and easing the workload of human administrators.

Generally, Kubernetes seems to be exploring its role as an platform for Stateful applications. It will be interesting to see where this soul-searching of the Kubernetes community will lead. Will Kubernetes decide to focus only on stateless, microservice-like workloads which is what it got famous for, or will it evolve into a platform for automating the deployment and lifecycle of Stateful applications? No matter the outcome, the journey will be interesting!

6.2 Future Work

In the near future, we would like to implement additional functionality for Day-2 actions in Cassandra Operator. More specifically, we would like to support:

- Backups & Restores
- A Web GUI for Better UX
- Automated Repairs by Cassandra Reaper
- Multi-Region Clusters

Many of these are proposals and are being actively design in the Rook project. [3]

We would also like to see better support for local disks in Kubernetes.

- Better Support for Local Disks: More specifically, Kubernetes should have a way to monitor disks and take actions in case of a failure. An interesting proposal in this direction is called ”Persistent Volume Monitor”. [19]
- Better Support for Multi-Region Clusters: Some projects in this direction include federation-v2[20] and crossplane[21].

Bibliography

[1] CoreOS, "Introducing operators: Putting operational knowledge into software," 2016.

[2] "State of containers - overview." https://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html.

[3] "rook - storage orchestration for kubernetes." https://github.com/rook/rook.

[4] "Ceph." https://ceph.com/.

[5] "Scylla - the real-time big data database." https://www.scylladb.com/.

[6] "Java management extensions." https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html.

[7] "Kubernetes services." https://kubernetes.io/docs/concepts/services-networking/service/.

[8] "Anti-entropy repairs in apache cassandra." http://cassandra.apache.org/doc/4.0/operating/repair.html.

[9] "Cassandra reaper - automated repairs for apache cassandra." https://github.com/thelastpickle/cassandra-reaper.

[10] "The go programming language." https://golang.org/.

[11] "Jetstack navigator." https://github.com/jetstack/navigator.

[12] "Mysql operator by oracle." https://github.com/oracle/mysql-operator.

[13] "Sample controller." https://github.com/kubernetes/sample-controller/.

[14] "Go client for kubernetes." https://github.com/kubernetes/client-go.

[15] "Cockroachdb operator for rook." https://github.com/rook/rook/pull/1777.

[16] "Kubernetes code generator." https://github.com/kubernetes/code-generator.

[17] "Cobra - a commander for modern go cli interactions." https://github.com/spf13/cobra.

[18] "go-nodetool, a golang library to manage cassandra / scylla clusters." https://github.com/yanniszark/go-nodetool.

[19] "Persistent volume monitoring proposal." https://github.com/kubernetes/community/pull/1484.

[20] "Kubernetes federation v2." https://github.com/kubernetes-sigs/federation-v2.

[21] "Crossplane - an open-source multi-cloud control plane." https://github.com/crossplaneio/crossplane.