Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Πρόσθετο Τόμων Container Storage Interface (CSI) για Ενιαίες Ιεραρχίες Συστημάτων Αρχείων Πολλών Terabytes σε Συστοιχίες Kubernetes με Τοπική Αποθήκευση Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Θωμαδάκης

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2024

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

Πρόσθετο Τόμων Container Storage Interface (CSI) για Ενιαίες Ιεραρχίες Συστημάτων Αρχείων Πολλών Terabytes σε Συστοιχίες Kubernetes με Τοπική Αποθήκευση Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Θωμαδάκης

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Μαρτίου 2024.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ	Αναπλ. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2024

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Container Storage Interface (CSI) Volume Plugin for Multi-Terabyte, Unified Filesystem Hierarchies over Local Storage Kubernetes Clusters

DIPLOMA THESIS

Georgios Thomadakis

Supervisor: Nectarios Koziris
Professor NTUA

Athens, February 2024

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Container Storage Interface (CSI) Volume Plugin for Multi-Terabyte, Unified Filesystem Hierarchies over Local Storage Kubernetes Clusters

DIPLOMA THESIS

Georgios Thomadakis

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the three-member examination committee on the 29th of March 2024.

. . . . .	. . . . .	. . . . .
Nectarios Koziris	Georgios Goumas	Dionisios Pnevmatikatos
Professor NTUA	Associate Professor NTUA	Professor NTUA

Athens, February 2024

. . . . . .

Γεώργιος Θωμαδάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Στη μητέρα μου

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.999992.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro, Myriad Pro και `Consolas`.

Περίληψη

Γρήγορη, τοπική αποθήκευση NVMe, σε συνδυασμό με προηγμένες υπηρεσίες δεδομένων, μέσω ενός ελαφρού, κρίσιμου μονοπατιού δεδομένων, είναι αυτό που πραγματικά επιταχύνει και ασφαλίζει τις σημερινές επιχειρηματικές stateful εφαρμογές.

Η διπλωματική εργασία αυτή προσπαθεί να αντιμετωπίσει μερικές από τις παραπάνω απαιτήσεις αποθήκευσης. Με κίνητρο την αυξανόμενη ζήτηση για συστήματα αποθήκευσης μεγάλης κλίμακας και υψηλής απόδοσης, αυτή η έρευνα προτείνει μια καινοτόμο λύση, σχεδιασμένη να συγκεντρώνει την αποθήκευση από διαφορετικές τοπικές συσκευές και κόμβους σε ενιαίους χώρους ονομάτων. Αυτοί οι χώροι ονομάτων εκτίθενται σε και χρησιμοποιούνται από containerized φορτία εργασίας ως μόνιμοι τόμοι εντός συστοιχιών Kubernetes μέσω του προτύπου Container Storage Interface (CSI).

Σε αυτή τη διατριβή, χρησιμοποιούμε το union σύστημα αρχείων MergerFS στο Kubernetes μέσω ενός πρόσθετου τόμων CSI για να συνδυάσουμε πολλαπλούς τοπικούς χώρους αποθήκευσης σε ένα σύστημα αρχείων βασισμένο στο FUSE. Συνεχίζουμε αξιολογώντας την απόδοσή του, συγκρίνοντάς την με υπάρχουσες λύσεις αποθήκευσης επιχειρηματικού επιπέδου, για να αποφανθούμε, εάν τελικά αποτελεί εφικτή επιλογή για την υποστήριξη μεγάλων, σε επίπεδο petabytes, συγκεντρώσεων αποθήκευσης στο νέφος.

Λέξεις-Κλειδιά

Container Storage Interface, Πρόσθετο Τόμων CSI, Οδηγός CSI, Kubernetes, Τοπική Αποθήκευση, Συγκέντρωση Αποθήκευσης, FUSE, Συστήματα Αρχείων Union, MergerFS

Abstract

Fast, local NVMe storage, coupled with advanced data services through a lightweight critical data path, is what truly accelerates and secures business-critical stateful applications today.

This diploma thesis endeavors to address some of these storage requirements. Motivated by the growing demand for large-scale and high-performance storage systems, this research proposes a novel solution designed to aggregate storage across different local devices and nodes into single, unified namespaces. These namespaces are exposed to and consumed by containerized workloads as persistent volumes within Kubernetes clusters through the Container Storage Interface (CSI) standard.

In this thesis, we employ the MergerFS union filesystem in Kubernetes through a CSI volume plugin to combine multiple local storage spaces into a FUSE-based filesystem. We proceed to assess its performance, comparing it with existing enterprise-grade storage solutions, to determine if it is a viable option for supporting large, petabyte-level, storage pools in the cloud.

Keywords

Container Storage Interface, CSI Volume Plugin, CSI Driver, Kubernetes, Local Storage, Storage Pooling, FUSE, Union Filesystems, MergerFS

Ευχαριστίες

Προτού προχωρήσουμε, σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της διπλωματικής εργασίας κ. Νεκτάριο Κοζύρη, ο οποίος καλλιέργησε μέσω των μαθημάτων του το ενδιαφέρον μου για τα Υπολογιστικά Συστήματα.

Επίσης, ευχαριστώ θερμά τον διδάκτορα κ. Βαγγέλη Κούκη, η στοχευμένη καθοδήγηση και βαθιά γνώση του οποίου ήταν καθοριστικές για τη πορεία και το αποτέλεσμα αυτής της εργασίας και καταλυτικές για τη διαμόρφωση του τρόπου σκέψης μου σαν μηχανικού και συνεργάτη. Ακόμα, θα ήθελα να τον ευχαριστήσω για την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία υπό την αιγίδα την εταιρείας Arrikto. Στην Arrikto είχα την τιμή να έρθω σε επαφή με μία ομάδα ταλαντούχων φοιτητών και ατόμων της εταιρείας, ώστε να εργαστούμε, τόσο ατομικά όσο και ομαδικά, πάνω σε πεδία και με εργαλεία που αποτελούν την αιχμή της cloud τεχνολογίας και εφαρμογών, σε ένα άκρως επαγγελματικό, ενθαρρυντικό, και φιλικό περιβάλλον.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για τη διαρκή υποστήριξη και αγάπη της όλα αυτά τα χρόνια σε κάθε μου βήμα και καθ'όλη τη διάρκεια των σπουδών μου.

Γεώργιος Θωμαδάκης
Ιανουάριος 2024

Contents

List of Figures

List of Algorithms

Εκτενής Ελληνική Περίληψη

Το ελληνικό κείμενο της παρούσας εργασίας δίνεται παρακάτω σε μορφή εκτενούς περίληψης.

1 Εισαγωγή

Σε αυτό το εναρκτήριο κεφάλαιο εισαγάγουμε το αντικείμενο της εργασίας.

1.1 Κίνητρο

Η αποθήκευση δεδομένων αποτελεί τη βάση των σύγχρονων αρχιτεκτονικών κέντρων δεδομένων και stateful εφαρμογών. Αποθήκευση υψηλής χωρητικότητας και απόδοσης, χαμηλής καθυστέρησης και με ανοχή σφαλμάτων είναι ένα κρίσιμο στοιχείο για υποδομές και επιχειρήσεις που διαχειρίζονται φορτία εργασίας υψίστης σημασίας με σημαντικές απαιτήσεις δεδομένων.

Οι παροχείς υπηρεσιών νέφους και αποθηκευτικού χώρου έχουν εξελιχθεί για να καλύψουν μια ευρεία ποικιλία απαιτήσεων υπολογιστικής ισχύος και αποθηκευτικών αναγκών, εγκαθιδρύοντας ταυτόχρονα μια εξαιρετικά κερδοφόρα αγορά. Την ίδια στιγμή, η ταχεία άνοδος των συστημάτων ενορχήστρωσης containers (container orchestrator systems/COs), με απόγειό τους το Kubernetes, έχει επαναπροσδιορίσει τον τρόπο ανάπτυξης και διαχείρισης containerized εφαρμογών. Καθώς οι οργανισμοί στρέφονται όλο και περισσότερο στη συσκευασία των εφαρμογών τους σε containers, όπου κάθε stateful εφαρμογή διαθέτει δυνητικά τη δικιά της αποθήκη δεδομένων, η ζήτηση για αποτελεσματικές και κλιμακώσιμες λύσεις αποθήκευσης εντός συστοιχιών Kubernetes έχει γίνει ζωτικής σημασίας.

Ωστόσο, είτε οι εταιρείες επιλέγουν υποδομές on-premises, είτε νέφους, κάνοντας χρήση του Kubernetes ή όχι, το αιώνιο δίλημμα ανάπτυξης εφαρμογών παραμένει: τοπική ή

κοινόχρηστη αποθήκευση; Η τοπική ή απευθείας συνδεδεμένη αποθήκευση προσφέρει αυξημένα IOPS και ρυθμαπόδοση, αλλά συνοδεύεται από μειωμένη ευελιξία και επεκτασιμότητα. Από την άλλη πλευρά, η κοινόχρηστη ή απομακρυσμένη αποθήκευση (π.χ., EBS στο Amazon Web Services, Azure Disk Storage, Persistent Disks στο Google Cloud Platform) είναι πραγματικά ευέλικτη, διευκολύνοντας την εφαρμογή snapshots, αντιγράφων ασφαλείας και μετακινήσεων μεταξύ κόμβων, αλλά περιορίζει την απόδοση και αυξάνει το κόστος.

Ακόμη και αν η τοπική αποθήκευση έχει το καθαρό προβάδισμα απόδοσης και τιμής και οι περισσότερες εφαρμογές θα επωφελούνταν σημαντικά αν εκτελούνταν σε αποδοτικούς, τοπικούς NVMe SSD, πολλές εταιρείες φαίνεται ότι είναι διστακτικές ή αδυνατούν να κάνουν το άλμα στην τοπική αποθήκευση επειδή εξαρτώνται πραγματικά από την ευελιξία της κοινόχρηστης αποθήκευσης. Το να έχει κανείς τα δεδομένα του ασφαλή, υψηλά διαθέσιμα και φορητά είναι κρίσιμο. Άλλο τόσο κρίσιμοι, όμως, είναι οι χρόνοι καθυστέρησης E/E στην κλίμακα των μικροδευτερολέπτων και οι μειωμένοι λογαριασμοί.

Είναι δυνατόν οι stateful εφαρμογές να εκτελούνται σε τοπικούς δίσκους NVMe, ενώ παράλληλα διατηρούν όλα τα πλεονεκτήματα της κοινόχρηστης αποθήκευσης; Είναι η τοπική και η κοινόχρηστη αποθήκευση εγγενώς αμοιβαία αποκλειόμενες; Πιστεύουμε ότι το παραπάνω δίλημμα δεν ισχύει πλέον σήμερα. Οι σύγχρονες εφαρμογές δεν θα πρέπει πλέον να ανταλλάσουν την απόδοση για την ευελιξία και αντίστροφα. Ένα σύγχρονο, επιχειρησιακό σύστημα αποθήκευσης και διαχείρισης δεδομένων, θα έπρεπε να εκμεταλλεύεται την τοπική αποθήκευση μέσω ενός ελαφρού, κρίσιμου μονοπατιού δεδομένων παρέχοντας παράλληλα προηγμένες δυνατότητες αποθήκευσης και υπηρεσίες δεδομένων στο νέφος, ακριβώς όπως και η κοινόχρηστη αποθήκευση. Ένα τέτοιο σύστημα θα επέτρεπε στους επιστήμονες δεδομένων και τους προγραμματιστές να προτυποποιούν, να δοκιμάζουν και να καταθέτουν γρήγορα τις εφαρμογές και τα μοντέλα μηχανικής μάθησής τους, και τις εταιρείες να ενισχύουν την απόδοσή τους, να μειώνουν τα κόστη αποθήκευσης και να ξεκλειδώνουν νέες επιχειρηματικές δυνατότητες.

Σε αυτή τη διατριβή, εμπνεόμενοι από αυτήν την ισχυρή πεποίθηση για μια κοινή λύση αποθήκευσης, ικανή να συνδυάσει τα καλύτερα και των δύο κόσμων, εξερευνούμε ένα κομμάτι αυτού του παζλ: την κλιμάκωση. Εξετάζουμε έναν απλό τρόπο συγχώνευσης

μεγάλων, πολλαπλών terabytes, μόνιμων αποθηκευτικών συγκεντρώσεων σε υψηλής απόδοσης τοπικούς δίσκους εντός συστοιχιών Kubernetes.

1.2 Διατύπωση Προβλήματος

Η τοπική και η κοινόχρηστη αποθήκευση αποτελούν τις δύο κυρίαρχες, συμπληρωματικές επιλογές πρόσβασης σε αποθήκευση. Η πρώτη παρέχει υψηλή απόδοση αλλά περιορισμένη ευελιξία, ενώ η δεύτερη είναι εξαιρετικά ευέλικτη, αλλά μπορεί να κάνει συμβιβασμούς ως προς την απόδοση.

Οι παροχείς υπηρεσιών νέφους προσφέρουν και τις δύο επιλογές αυτές μαζί με ενδιάμεσες υβριδικές λύσεις. Ο κανόνας είναι, ότι η τοπική αποθήκευση λειτουργεί ως η οικονομικά συμφέρουσα, αρχικού επιπέδου βαθμίδα, ενώ η επεκτασιμότητα της κοινόχρηστης αποθήκευσης ξεκλειδώνεται μέσω ανώτερων βαθμίδων και της αντίστοιχης αύξησης της τιμής.

Τώρα, ας λάβουμε υπόψη ένα σενάριο, όπου μια ομάδα προγραμματιστών τρέχει μία συστοιχία Kubernetes στο νέφος. Έχουν επιλέξει να εκμεταλλευτούν την τοπική αποθήκευση μπλοκ που συνδέεται σε κάθε κόμβο στη συστοιχία, για να επωφεληθούν από την αυξημένη απόδοση σε IOPS και τη χαμηλή καθυστέρηση, χρησιμοποιώντας ένα κατάλληλο πρόσθετο μόνιμων τόμων.

Ωστόσο, τι συμβαίνει όταν τα δεδομένα της εφαρμογής αυξάνονται και εξαντλείται η χωρητικότητα σε έναν ή περισσότερους κόμβους; Ποιες επιλογές υπάρχουν; Μία επιλογή είναι η μετάβαση στη λύση κοινόχρηστης αποθήκευσης του παροχέα νέφους για κλιμάκωση προς τα έξω (scale-out) συνδέοντας μία συσκευή αποθήκευσης από μία εξωτερική βάση. Ενώ οι παρεχόμενες προνομιακές βαθμίδες μπορεί να προσφέρουν επαρκή απόδοση, έρχονται με αυξημένο κόστος. Μία άλλη επιλογή είναι η αναβάθμιση ενός υποσυνόλου των κόμβων σε συσκευές τοπικής αποθήκευσης υψηλότερης χωρητικότητας, σε μία προσπάθεια προσομοίωσης αρχιτεκτονικής κλιμάκωσης προς τα πάνω (scale-up). Ωστόσο, η μετακίνηση ολόκληρης της εφαρμογής και των δεδομένων της μπορεί να είναι δύσκολη, κουραστική και καμιά φορά έως και αδύνατη, λόγω των περιορισμών της τοπικής αποθήκευσης.

Την ίδια στιγμή, ο υπόλοιπος αποθηκευτικός χώρος των κόμβων εντός της συστοιχίας μπορεί απλά να “κάθεται” εκεί, απολύτως ελεύθερος και διαθέσιμος.

Υπάρχει η δυνατότητα να συνδυαστούν οι αποθηκευτικοί χώροι των κόμβων της συστοιχίας σε μια ενιαία ιεραρχία, ουσιαστικά φέρνοντας όλους τους χώρους αποθήκευσης, που είναι διαθέσιμοι σε κάθε μεμονωμένο κόμβο, σε μία μοναδική ”δεξαμενή” αποθήκευσης και παρουσιάζοντάς την στον χρήστη ως μία ενιαία, συνεκτική όψη;

Σε αυτή τη διατριβή παρουσιάζουμε έναν μηχανισμό για τη συγχώνευση των χώρων αποθήκευσης από διάφορες συσκευές και κόμβους στη συστοιχία για την παροχή τόμων, που επιτυγχάνουν χωρητικότητες που υπερβαίνουν την διαθέσιμη χωρητικότητα μεμονωμένων κόμβων.

1.3 Υπάρχουσες Λύσεις

Υπάρχουν αρκετά εξεζητημένα και ώριμα συστήματα και πλατφόρμες αποθήκευσης, που είναι ικανά να συγκεντρώνουν ολόκληρη τον αποθηκευτικό χώρο της συστοιχίας σε μία ενιαία “δεξαμενή” αποθήκευσης, συνοδευόμενα από προηγμένες δυνατότητες όπως, αντιγραφή δεδομένων (replication), κατανεμημένη αποθήκευση, δημιουργία snapshots και άλλα.

Ένα παράδειγμα που χρησιμοποιείται ευρέως είναι το GlusterFS, που προσφέρει κλιμακώσιμα, κατανεμημένα συστήματα αρχείων δικτύου σε εξοπλισμό εμπορίου, ενώ είναι δωρεάν και ανοιχτού κώδικα. Το GlusterFS συγκεντρώνει τους αποθηκευτικούς πόρους δίσκων από πολλούς απομακρυσμένους διακομιστές σε μοναδικούς προσαρτώμενους τόμους. Οι διακομιστές προσπελαύνονται μέσω Ethernet ή InfiniBand, και οι πελάτες μπορούν να προσαρτήσουν έναν τόμο GlusterFS μέσω της διεπαφής FUSE και των πρωτοκόλλων NFS ή SMB.

Το GlusterFS παρέχει διάφορους τύπους όγκων, που καθορίζουν πώς τα αρχεία και τα δεδομένα δρομολογούνται στους υποκείμενους διακομιστές. Από προεπιλογή, τα αρχεία διανέμονται σε διάφορους διακομιστές χωρίς αντιγραφή για εξοικονόμηση χώρου αποθήκευσης. Για μείωση του κινδύνου απώλειας δεδομένων, οι πελάτες μπορούν να επιλέξουν *αντιγραμμένους* τόμους, όπου ακριβή αντίγραφα όλων των αρχείων αποθηκεύονται σε όλους τους διακομιστές. Επιπλέον, το GlusterFS προσφέρει *διανεμημένους αντιγραμμένους τόμους*, που συνδυάζουν τόσο τη διανομή, όσο και την αντιγραφή αρχείων, καθώς και *διασκορπισμένους (dispersed) τόμους* και *διανεμημένους διασκορπισμένους τόμους*.

Το GlusterFS έχει εξαγοραστεί από τη Red Hat και ενσωματώνεται στο Kubernetes μέσω του Container Storage Interface (CSI) χάρις στον οδηγό CSI του. Ωστόσο, αξίζει να σημειωθεί ότι το Red Hat Gluster Storage, όπως έχει μετονομαστεί από τη Red Hat, βρίσκεται στην τελική φάση του κύκλου ζωής του, με ημερομηνία λήξης της υποστήριξής του την 31η Δεκεμβρίου 2024. Επιπλέον, ο οδηγός CSI του GlusterFS δεν υποστηρίζεται πλέον και προορίζεται για απόσυρση, ενώ το ενσωματωμένο στο Kubernetes πρόσθετο τόμων GlusterFS έχει αφαιρεθεί από την έκδοση v1.26 [1] και έπειτα.

Τελειώνοντας, ας σημειωθεί ότι ένα επαναλαμβανόμενο χαρακτηριστικό που παρατηρείται σε πολλά από τα σύγχρονα συστήματα αποθήκευσης, συμπεριλαμβανομένων και συστημάτων αιχμής που χρησιμοποιούν τοπική αποθήκευση, είναι η εισαγωγή υπερβολικού "φορτίου". Αυτά τα συστήματα τείνουν να περιλαμβάνουν πληθώρα και περίπλοκα επίπεδα αφαίρεσης μεταξύ της εφαρμογής και των υποκείμενων δίσκων για την υλοποίηση των επιθυμητών υπηρεσιών δεδομένων, όπως ικανότητες λήψης snapshots ή αντιγραφή δεδομένων. Τα επίπεδα αφαίρεσης εντός του κρίσιμου μονοπατιού δεδομένων οδηγούν αναπόφευκτα σε αυξημένη καθυστέρηση και επιβράδυνση της απόδοσης σε E/E.

1.4 Προτεινόμενη Λύση

Όπως έχει περιγραφεί μέχρι στιγμής, σε αυτή τη διατριβή παρουσιάζουμε και εξηγούμε έναν απλό proof-of-concept μηχανισμό για τη συγχώνευση μικρότερων αποθηκευτικών πόρων σε έναν ενιαίο χώρο ονομάτων αποθήκευσης που παρουσιάζεται ως μόνιμοι τόμοι σε containerized φορτία εργασίας.

Η προτεινόμενή μας λύση χρησιμοποιεί συστήματα αρχείων union, συγκεκριμένα την υλοποίηση του MergerFS, καταξιωμένου για τα ποικίλα χαρακτηριστικά του και τη δρομολόγηση βάση πολιτικής. Συγκεντρώνει τους τόμους αποθήκευσης από διάφορους δίσκους και κόμβους μέσω Ethernet, συγχωνεύοντάς τους για να παρέχει μία προσάρτηση union (union mount) σε container καταναλωτές. Ανεπτυγμένο αποκλειστικά για συστοιχίες Kubernetes, λειτουργεί ως πρόσθετο τόμων μέσω του Container Storage Interface (CSI). Το αποκαλούμε Union CSI.

[1] https://kubernetes.io/blog/2022/09/26/storage-in-tree-to-csi-migration-status-update-1.25/#timeline-and-status

Ένα σημαντικό στοιχείο σχεδιασμού του είναι, ότι το Union CSI δεν αποτελεί ένα αυτόνομο σύστημα αποθήκευσης, αλλά μια επέκταση ή περιτύλιγμα για υπάρχοντα συστήματα αποθήκευσης. Σε περιπτώσεις όπου η ζήτηση από έναν χρήστη για αποθήκευση από το αγαπημένο του είδος αποθήκευσης δεν μπορεί να ικανοποιηθεί, επειδή κανένας δίσκος που συνδέεται στους κόμβους της συστοιχίας δεν μπορεί από μόνος του να καλύψει τη ζητούμενη χωρητικότητα, το Union CSI μπορεί να βοηθήσει. Το Union CSI διαιρεί το αρχικό αίτημα του χρήστη για ”πολύ μεγάλη” αποθήκευση σε πολλαπλά μικρότερα και εφικτά αιτήματα, που ανακατευθύνονται στο υποκείμενο σύστημα αποθήκευσης, προσφέροντας μία προσάρτηση union των παρεχόμενων τόμων του. Οι συγκεκριμένες απαιτήσεις για τα υποκείμενα συστήματα αποθήκευσης αναλύονται στην Ενότητα 2.1. Ωστόσο, το Union CSI προορίζεται να συνδυάζεται με συστήματα που χρησιμοποιούν αποτελεσματικά την τοπική αποθήκευση και να εγκαθίσταται σε συστοιχίες με γρήγορους, τοπικούς δίσκους NVMe, επιτρέποντας στο Union CSI να τοποθετήσει το MergerFS πάνω από τόμους υψηλής απόδοσης.

Ο στόχος είναι να επιτρέπεται στους χρήστες να ενσωματώνουν το Union CSI με το σύστημα τοπικής αποθήκευσης της προτίμησής τους για να αξιοποιήσουν ενοποιημένους τόμους που υπερβαίνουν τη χωρητικότητα οποιουδήποτε διαθέσιμου μοναδικού δίσκου στη συστοιχία. Αυτό υπόσχεται σημαντικές δυνατότητες κλιμάκωσης προς τα έξω και προς τα πάνω. Είτε αυξάνεται η χωρητικότητα ενός δίσκου, είτε προστίθεται στη συστοιχία ένας νέος κόμβος με τοπική αποθήκευση, μπορούν όλοι να ενσωματωθούν στη ”δεξαμενή” αποθήκευσης του Union CSI για κατανάλωση.

2 Σχεδίαση

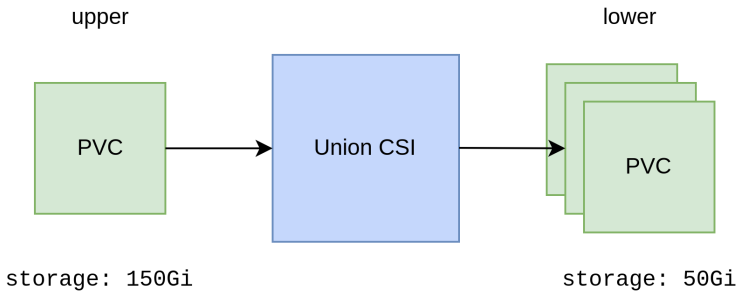
Σε αυτό το κεφάλαιο παρέχουμε τις σχεδιαστικές αποφάσεις και τους μηχανισμούς που απαρτίζουν την εφαρμογή λογισμικού αυτής της διατριβής.

2.1 Επισκόπηση

Στοχεύουμε να εισαγάγουμε ένα πρόσθετο τόμων στο Kubernetes, για την παροχή τόμων συστημάτων αρχείων union, συνδυάζοντας τοπικούς τόμους αποθήκευσης. Αυτό το πρόσθετο ενσωματώνεται στο Kubernetes μέσω του Container Storage Interface (CSI) και του οδηγού CSI του. Το αποκαλούμε Union CSI.

Προκειμένου να αναπτύξουμε γρήγορα ένα απλό, λειτουργικό παράδειγμα και να υλοποιήσουμε την ιδέα μας, αποφασίσαμε να εκμεταλλευτούμε και να ενσωματώσουμε υπάρχουσες λύσεις και δομές. Η βασική ιδέα είναι ότι το Union CSI, στην παρούσα μορφή του, λειτουργεί ως ”μέτα-πρόσθετο”: ένα πρόσθετο που χρησιμοποιεί άλλα πρόσθετα για να συνδυάζει τους τόμους που προσφέρουν. Το Union CSI στοχεύει στην εγκατάστασή του στο Kubernetes και την απρόσκοπτη ενσωμάτωσή του με ένα υπάρχον σύστημα αποθήκευσης (εφόσον πληροί κάποιες προϋποθέσεις που θεσπίζουμε σε αυτό το κεφάλαιο), επιτρέποντας προσαρτήσεις union από τα στιγμιότυπα συστημάτων αρχείων του. Αυτή η προσέγγιση, όχι μόνο μας εξοικονομεί χρόνο και προσπάθεια, ιδιαίτερα στο πλαίσιο αυτής της εργασίας, αλλά επίσης εξαλείφει την ανάγκη για την επανεφεύρεση του τροχού, υλοποιώντας μια λύση βασισμένη στο LVM, ή παρόμοια, από το μηδέν.

Το Union CSI προκαλεί τη δημιουργία και διαγραφή τόμων από το κατώτερο σύστημα αποθήκευσης μέσω του PersistentVolumeClaim API του Kubernetes. Οι χρήστες του Kubernetes θα πρέπει να μπορούν να δημιουργούν ”πολύ μεγάλα” PVCs (όπου ”πολύ μεγάλα” σημαίνει ότι το ζητούμενο μέγεθος αποθήκευσης υπερβαίνει σημαντικά τον διαθέσιμο χώρο δίσκου κάθε μεμονωμένου κόμβου στη συστοιχία). Το Union CSI δημιουργεί εσωτερικά ένα σύνολο μικρότερων PVCs, το άθροισμα των οποίων ικανοποιεί την αρχική ζητούμενη χωρητικότητα. Οι ζητούμενα μικρότεροι τόμοι απευθύνονται στο κατώτερο σύστημα αποθήκευσης και μπορούν να χωρέσουν σε τοπικούς δίσκους σε διάφορους κόμβους.



Σχήμα 2.1: *Το Union CSI χειρίζεται σύνολα από PVCs*

Μία ακόμα ευθύνη που σκοπεύουμε να μεταθέσουμε από τη δική μας μεριά στο υποκείμενο πρόσθετο τόμων, είναι η δυνατότητα απομακρυσμένης πρόσβασης αποθήκευσης. Εάν το πρόσθετο τόμων, που χρησιμοποιείται από το Union CSI, αξιοποιεί τοπική αποθήκευση από τα μηχανήματα των κόμβων αντί για κοινόχρηστη αποθήκευση (σκεφτείτε EBS τόμους του AWS), τότε, θα πρέπει επίσης να είναι ικανό να συνδέει την αποθήκευση μεταξύ των κόμβων, δημιουργώντας ένα δίκτυο αποθήκευσης (SAN). Ωστόσο, αυτή η απαίτηση περιορίζει τις επιλογές μας για συστήματα αποθήκευσης που μπορούμε να χρησιμοποιήσουμε, μόνο σε αυτά που διαθέτουν δυνατότητες δικτύωσης αποθήκευσης (όπως μια υλοποίηση iSCSI) για την πρόσβαση σε απομακρυσμένους τόμους και τη μεταφορά δεδομένων από κόμβο σε κόμβο. Ενώ υπάρχουν κάποια τέτοια πρόσθετα, και εγκαθιστούμε ένα από αυτά και το παρουσιάζουμε σε επόμενο κεφάλαιο, έχει νόημα το ίδιο το Union CSI να υλοποιήσει αυτήν τη λειτουργία, καθώς αποτελεί απαραίτητο στοιχείο για την αξιοποίηση και συνδυασμό της τοπικής αποθήκευσης από διαφορετικούς κόμβους. Σίγουρα μία ακόμη προσθήκη στη λίστα μελλοντικών εργασιών μας.

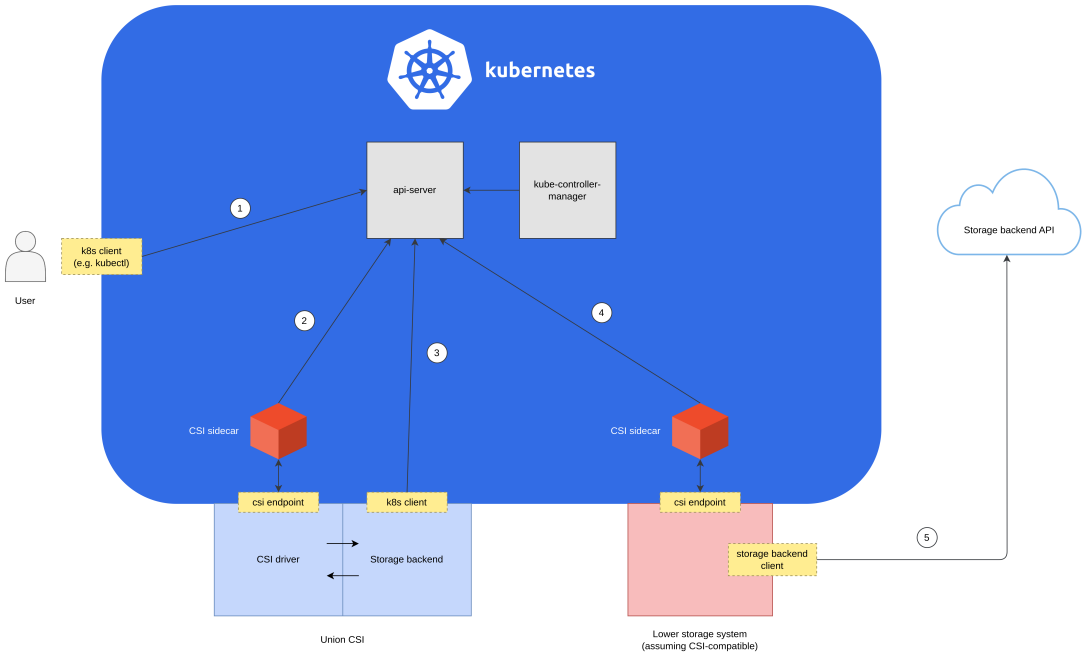
Το Union CSI δεν διαθέτει γνώση της κατάστασης, των λειτουργιών ή της ύπαρξης του υποκείμενου πρόσθετου. Το Union CSI κατευθύνεται προς ένα συγκεκριμένο σύστημα αποθήκευσης μέσω των παραμέτρων χρήστη που περνούν στο `CreateVolume` CSI RPC. Αυτό σημαίνει, ότι το ίδιο στιγμιότυπο του Union CSI μπορεί να ρυθμιστεί και να συνδυαστεί με διαφορετικές υπάρχουσες λύσεις αποθήκευσης εντός της ίδιας συστοιχίας.

Το Union CSI δεν επικοινωνεί απευθείας με το υποκείμενο πρόσθετο. Η επικοινωνία διαμεσολαβείται μέσω του Kubernetes. Το Union CSI χρησιμοποιεί το ίδιο API του Kubernetes που θα χρησιμοποιούσε ένας χρήστης (π.χ., PVCs και Pods) για να προκαλέσει τις επιθυμητές λειτουργίες τόμων του υποκείμενου πρόσθετου και παρακολουθεί τα αποτελέσματα μέσω ενημερώσεων στα αντικείμενα του API.

Το Union CSI χρησιμοποιεί το MergerFS για να ενώσει τους τόμους που παρέχονται από το υποκείμενο σύστημα αποθήκευσης και παρέχει ένα σύστημα αρχείων FUSE σε φορτία εργασίας. Οι υποκείμενοι τόμοι είτε προσαρτώνται τοπικά στον κόμβο όπου θα τρέξει μια εργασία, είτε απομακρυσμένα μέσω του δικτύου, και σε κάθε περίπτωση είναι υπό τη διαχείριση του συστήματος αποθήκευσής τους. Η δρομολόγηση των αρχείων και των δεδομένων που εγγράφονται στο σύστημα αρχείων FUSE μεταξύ των υποκείμενων τόμων, καθορίζεται από την επιλεγμένη πολιτική του MergerFS. Έτσι, το Union CSI μπορεί να υποστηρίξει διανεμημένη αποθήκευση, αλλά όχι αποθήκευση με αντίγραφα ή κορδονοειδή (stripped) αποθήκευση. Το MergerFS δεν διαμερίζει τα δεδομένα στα υποκείμενα συστήματα αρχείων.

Καθώς το Union CSI εξαρτάται άμεσα από το MergerFS για το μονοπάτι δεδομένων του, τα ίδια ”πρέπει και δεν πρέπει” που ισχύουν για προσαρτήσεις MergerFS ισχύουν και για τους τόμους Union CSI. Για μερικά παραδείγματα από τα ”δεν πρέπει” παρακαλούμε αναφερθείτε στο αρχείο `README.md` στο αποθετήριο του MergerFS στο GitHub [1].

[1] https://github.com/trapexit/mergerfs?tab=readme-ov-file#what-should-mergerfs-not-be-used-for



1 A user makes a request to the Kubernetes API server for a storage operation from the Union CSI plugin and Kubernetes updates the cluster state accordingly. For example, a user creates a PVC using a Union CSI StorageClass and the PersistentVolume controller of kube-controller-manager updates the PVC for an external provisioner.

2 A Kubernetes CSI sidecar container deployed with a Union CSI instance picks up the updated state of Kubernetes and invokes the corresponding RPC of the Union CSI plugin via a gRPC endpoint. For example, the external-provisioner sidecar container detects the updated PVC and invokes the CreateVolume RPC.

3 The Union CSI plugin instance validates the RPC request (CSI driver component) and invokes the Kubernetes API (storage backend component) to retrieve the current state of the requested storage operation and update it if necessary. For example, Union CSI creates a sets of PVCs using a StorageClass of the lower storage system.

4 A Kubernetes CSI sidecar container deployed with a lower storage system instance picks up the updated state of Kubernetes and invokes the corresponding RPC of the lower storage system via a gRPC endpoint. For example, the external-provisioner sidecar container detects one or more PVCs and invokes the CreateVolume RPC. We assume that the storage system integrates with Kubernetes through CSI.

5 The storage system carries out the requested storage operation. For example, the storage system provisions the requested volume(s) on behalf of Union CSI. The storage system (likely) invokes its storage backend to retrieve and update its state, which can be an external API or an extended Kubernetes API.

Σχήμα 2.2: *Επισκόπηση σχεδίασης του Union CSI μέτα-πρόσθετου*

2.2 Ορολογία

Term	Description
Πρόσθετο Τόμων CSI	Ένα συμβατό με το CSI πρόσθετο τόμων από εξωτερικό πάροχο που προσφέρει αποθήκευση σε containerized φορτία εργασίας που εκτελούνται σε περιβάλλον CO μέσω των υπηρεσιών CSI. Σημείωση: Στο Kubernetes, ο όρος ”CSI volume plugin” χρησιμοποιείται περιστασιακά για να αναφερθεί στον εσωτερικό μηχανισμό CSI που χρησιμοποιεί το Kubernetes για την αλληλεπίδραση και ενσωμάτωση με εξωτερικούς ”CSI volume drivers”.
Οδηγός Τόμων CSI	Το τμήμα ενός πρόσθετου τόμων CSI που υλοποιεί τις υπηρεσίες CSI μέσω ενός σημείου gRPC.
Άνω/Ανώτερο	Επίθετο για οντότητες, στοιχεία και πόρους που σχετίζονται με το πρόσθετο τόμων Union CSI, π.χ. άνω πρόσθετο τόμων, άνω τόμος, άνω PVC, άνω PV, κλπ.
Κάτω/Κατώτερο	Επίθετο για οντότητες, στοιχεία και πόρους που σχετίζονται με το υποκείμενο σύστημα αποθήκευσης που χρησιμοποιείται από το πρόσθετο Union CSI, π.χ. κατώτερο πρόσθετο τόμων, κατώτερος τόμος, κάτω PVC, κάτω PV, κλπ.

Από εδώ και στο εξής, ολόκληρο το λογισμικό του Union CSI θα αναφέρεται ως *πρόσθετο τόμων Union CSI*, *πρόσθετο Union CSI* ή απλά *Union CSI*. Το συγκεκριμένο στοιχείο που υλοποιεί και παρέχει τις υπηρεσίες CSI (οδηγός CSI), και είναι υποσύνολο του πρόσθετου τόμων Union CSI, θα αναφέρεται ως *οδηγός τόμων Union CSI* ή απλά *οδηγός Union CSI*. Θα χρησιμοποιήσουμε επίσης τους όρους ”άνω”/”ανώτερο” και ”κάτω”/”κατώτερο” για να μας βοηθήσουν να διακρίνουμε μεταξύ του Union CSI και των πρόσθετων που χρησιμοποιεί.

2.3 Στόχοι

Σχετικά με τις λειτουργίες και το διαχωρισμό αρμοδιοτήτων μεταξύ του Union CSI (άνω πρόσθετο) και του υποκείμενου συστήματος αποθήκευσης (κάτω πρόσθετο), το Union CSI **θα**:

- Δημιουργεί και διαγράφει PersistentVolumeClaims που χρησιμοποιούν ένα StorageClass του κατώτερου συστήματος αποθήκευσης για να προκαλέσει την παροχή και τη διαγραφή των παρεχόμενων τόμων του.
- Δημιουργεί και διαγράφει Pods που χρησιμοποιούν τα PersistentVolumeClaim που δημιούργησε το Union CSI για το κατώτερο σύστημα αποθήκευσης, προκειμένου να προκαλέσει τη σύνδεση/αποσύνδεση και την προσάρτηση/αποπροσάρτηση των παρεχόμενων τόμων του.
- Χρησιμοποιεί το MergerFS μέσα στο container του Pod για να συγχωνεύσει τους τόμους του κατώτερου συστήματος αποθήκευσης, που έχουν προσαρτηθεί στο container, και προσαρτά το προκύπτον σύστημα αρχείων union πίσω στον κόμβο οικοδεσπότη, υπό μορφή καταλόγου.
- Προσαρτά (bind) το σύστημα αρχείων union του κόμβου οικοδεσπότη στο καθορισμένο από το Kubelet μονοπάτι, για να προσαρτηθεί περαιτέρω εντός των container καταναλωτών.

2.4 Μη-Στόχοι

Σχετικά με τις λειτουργίες και το διαχωρισμό αρμοδιοτήτων μεταξύ του Union CSI (άνω πρόσθετο) και του υποκείμενου συστήματος αποθήκευσης (κάτω πρόσθετο), το Union CSI **δεν θα**:

- Αναλαμβάνει άμεσα δέσμευση, διαχείριση ή απελευθέρωση αποθηκευτικού χώρου μπλοκ από τους δίσκους, που είναι συνδεδεμένοι στους κόμβους της συστοιχίας. Το Union CSI βασίζεται στο κατώτερο σύστημα αποθήκευσης για αυτήν τη λειτουργικότητα.
- Δημιουργεί, χρησιμοποιεί ή ανιχνεύει ένα SAN οποιουδήποτε είδους στη συστοιχία. Το Union CSI βασίζεται στο κατώτερο σύστημα αποθήκευσης για αυτήν τη

λειτουργικότητα. Εάν το κατώτερο σύστημα αποθήκευσης δεν είναι σε θέση να έχει πρόσβαση στους τόμους του από διαφορετικούς κόμβους της συστοιχίας, το Union CSI θα αποτύχει να χρονοδρομολογήσει ένα Pod που χρησιμοποιεί τόμους από διαφορετικούς κόμβους.

- Επικοινωνεί άμεσα με το κατώτερο σύστημα αποθήκευσης μέσω ενός API. Το Union CSI θα δημιουργεί, θα παρακολουθεί και θα διαγράφει πόρους του Kubernetes (π.χ. PVCs, Pods) που προκαλούν τις επιθυμητές λειτουργίες τόμων του κατώτερου συστήματος αποθήκευσης.

2.5 Λεπτομέρειες Σχεδίασης

Σε αυτή την ενότητα, εξετάζουμε λεπτομερώς το πρότυπο CSI, εξηγώντας πώς ο οδηγός Union CSI υλοποιεί το CSI και επικεντρωνόμαστε στα RPCs που υποστηρίζει. Αναλύουμε πώς το backend του Union CSI εκμεταλλεύεται το API του Kubernetes, χρησιμοποιώντας τόσο ενσωματωμένους όσο και ειδικούς πόρους, για να προχωρήσει στη δημιουργία, διατήρηση, ανάκτηση, διαγραφή, σύνδεση, αποσύνδεση, προσάρτηση και αποπροσάρτηση των κατώτερων και ανώτερων τόμων. Επιπλέον, καλύπτουμε πώς ένα containerized MergerFS πρόγραμμα μπορεί να συνδυάζει μικρότερους τόμους για να δημιουργήσει και να προσαρτήσει έναν Union CSI τόμο σε έναν κόμβο.

2.5.1 Διεπαφή RPC

Ο οδηγός Union CSI υλοποιεί τα ακόλουθα CSI RPCs:

Υπηρεσία RPC	Υποστηριζόμενα RPCs

Identity service	`GetPluginInfo`
	`Probe`
	`GetPluginCapabilities`

Controller service	`CreateVolume`
	`DeleteVolume`
	`ControllerPublishVolume`
	`ControllerUnpublishVolume`
	`ControllerGetCapabilities`

Node service	`NodePublishVolume`
	`NodeUnpublishVolume`
	`NodeGetInfo`
	`NodeGetCapabilities`

Πίνακας 2.1: *Τα υλοποιημένα RPCs του Union CSI*

Το Union CSI δεν υποστηρίζει και δεν υλοποιεί τα RPCs `NodeStageVolume` /`NodeUnstageVolume`. Ένας Union CSI τόμος (μία προσάρτηση MergerFS) συνδέεται και στη συνέχεια προσαρτάται σε ένα Node κατά τη διάρκεια του RPC `ControllerPublishVolume`. Ο τόμος ανεβαίνει στον κατάλογο `/var/lib/union-csi` από το Union CSI στον κόμβο, για να προσαρτηθεί (bind-mount) εντός ενός container Pod κατά τη διάρκεια του RPC `NodePublishVolume`.

`GetPluginInfo`: Επιστρέφει το όνομα του οδηγού Union CSI, `union.csi.union.io`, και την έκδοσή του, `dev`. Ένα StorageClass που αναφέρεται στο πρόσθετο Union CSI θα έχει αυτό το όνομα στο πεδίο `provisioner`.

`Probe`: Επιστρέφει ένα κενό `ProbeResponse`, δείχνοντας ότι το συγκεκριμένο πρόσθετο είναι έτοιμο.

`GetPluginCapabilities`: Επιστρέφει τη δυνατότητα πρόσθετου `CONTROLLER_SERVICE` (το Union CSI υλοποιεί τα RPCs της υπηρεσίας Controller).

`CreateVolume`: Δημιουργεί τα κατώτερα PersistentVolumeClaims που απευθύνονται στο κατώτερο πρόσθετο τόμων. Επιστρέφει το αναγνωριστικό τόμου του `CreateVolumeRequest.name` στο `CreateVolumeResponse.volume.volume_id`.

`DeleteVolume`: Διαγράφει τα κατώτερα PersistentVolumeClaims που δημιουργήθηκαν στο `CreateVolume`. Επιστρέφει ένα κενό `DeleteVolumeResponse`.

`ControllerPublishVolume`: Δημιουργεί ένα Pod που χρησιμοποιεί τα κατώτερα PersistentVolumeClaims και το αναθέτει στον καθορισμένο κόμβο. Το MergerFS container του Pod συγχωνεύει τους κατώτερους τόμους και προσαρτά το σύστημα αρχείων MergerFS σε έναν τόμο `hostPath` προσαρτημένο με αμφίδρομη προώθηση. Το σύστημα αρχείων προωθείται στον κόμβο στον κατάλογο `/var/lib/union-csi/volumes/<volume_id>/merged`. Επιστρέφει ένα κενό `ControllerPublishVolumeResponse`.

`ControllerUnpublishVolume`: Διαγράφει το Pod που χρησιμοποιεί τα κατώτερα PersistentVolumeClaims. Το MergerFS container αποπροσαρτά το σύστημα αρχείων πριν τον τερματισμό του. Επιστρέφει ένα κενό `ControllerUnpublishVolumeResponse`.

`ControllerGetCapabilities`: Επιστρέφει τις δυνατότητες χειριστή `CREATE_DELETE_VOLUME` και `PUBLISH_UNPUBLISH_VOLUME`.

`NodePublishVolume`: Προσαρτά (bind-mount) το σύστημα αρχείων MergerFS που βρίσκεται στο μονοπάτι `/var/lib/union-csi/volumes/<volume_id>/merged` στο καθορισμένο μονοπάτι-στόχο. Επιστρέφει ένα κενό `NodePublishVolumeResponse`.

`NodeUnpublishVolume`: Αποπροσαρτά το σύστημα αρχείων MergerFS από το καθορισμένο μονοπάτι-στόχο. Επιστρέφει ένα κενό `NodeUnpublishVolumeResponse`.

`NodeGetInfo`: Επιστρέφει το αναγνωριστικό κόμβου στο `NetGetInfoResponse.node_id`. Είναι το όνομα του αντικειμένου Node, στο οποίο εκτελείται το στιγμιότυπο του οδηγού. Ένα container του Union CSI που εκτελείται σε έναν κόμβο ανακτά το όνομα του κόμβου μέσω της μεταβλητής περιβάλλοντος `NODE_NAME`, που ορίζεται στο αρχείο δήλωσης του Pod στην τιμή του `spec.nodeName`.

`NodeGetCapabilities`: Επιστρέφει ένα κενό `NodeGetCapabilitiesResponse`, καθώς δεν υπάρχουν υποστηριζόμενες δυνατότητες κόμβου για αναφορά.

Συνεπώς, από δω και πέρα, θα χρησιμοποιούμε τον όρο *"Union CSI Controller"* για να αναφερθούμε σε ένα στιγμιότυπο του πρόσθετου Union CSI που υλοποιεί την υπηρεσία Controller και *"Union CSI Node"* για ένα στιγμιότυπο που υλοποιεί την υπηρεσία Node.

2.5.2 Διαίρεση Χωρητικότητας και Χρονοδρομολόγηση Τόμων

Ιδανικά, το Union CSI θα είχε έναν μηχανισμό για την παρακολούθηση της χρήσης του χώρου των δίσκων σε κάθε κόμβο, όπου εγκαθίσταται ένα στιγμιότυπο *Union CSI Node*. Αυτός ο μηχανισμός θα επέτρεπε στο Union CSI να καταγράφει τον συνολικό, τον χρησιμοποιημένο και τον διαθέσιμο χώρο αποθήκευσης και να λαμβάνει ενημερωμένες αποφάσεις σχετικά με τον κατάλληλο διαχωρισμό μεγέθους. Ωστόσο, η σχεδίαση και η υλοποίηση ενός τέτοιου μηχανισμού θα ήταν περίπλοκη. Επιπλέον, θα απαιτούσε από το Union CSI να συγχρονίζεται και να ακολουθεί τα αποθέματα αποθηκευτικού χώρου που διαχειρίζεται το κατώτερο σύστημα αποθήκευσης, προκειμένου να διασφαλίσει, ότι τα δύο πρόσθετα μοιράζονται μια κοινή εικόνα της χρήσης αποθηκευτικού χώρου. Διαφορετικά, οποιεσδήποτε προσπάθειες για παρακολούθηση της χρήσης χώρου και έξυπνων διαχωρισμών που θα πραγματοποιεί το Union CSI, θα ήταν ασύμφωνες και, εν τέλει, άκαρπες. Αυτή η προσέγγιση δεν είναι μια βιώσιμη επιλογή.

Ως αποτέλεσμα, επιλέξαμε μια πιο απλοϊκή προσέγγιση. Ανεξαρτήτως του ζητούμενου μεγέθους, το Union CSI προχωρά με έναν ”τυφλό” διαχωρισμό. Για παράδειγμα, ένα PVC του χρήστη, που δηλώνει έναν τόμο 100GiB με το Union CSI, οδηγεί πάντα σε δύο κατώτερα PVCs των 50GiB που δημιουργεί το πρόσθετο, ανεξαρτήτως της διαθεσιμότητας του αποθηκευτικού χώρου. Είναι ευθύνη του κατώτερου πρόσθετου να ελέγξει τη διαθεσιμότητα και να χρονοδρομολογήσει τους κατώτερους τόμους, αν είναι δυνατόν. Μια πιο ευέλικτη ρύθμιση θα παρέχονταν από έναν τόμο ConfigMap στο *Union CSI Controller* Pod που καθορίζει τον αριθμό των αντιγράφων PVCs που θα δημιουργηθούν και, συνεπώς, το τρέχον κομμάτι διαμέρισης. Το *Union CSI Controller* θα μπορούσε να παρακολουθεί το ConfigMap για ενημερώσεις, επιτρέποντας στον χρήστη να καθορίζει διαχωρισμούς μεγαλύτερης ή μικρότερης ευκρίνειας κατά τη διάρκεια της εκτέλεσης, σε περίπτωση που κάποιοι από τους δίσκους εξαντλούνται ή προστίθενται νέοι δίσκοι ή κόμβοι στη συστοιχία.

Ωστόσο, ακόμη και με ρυθμιζόμενα μεγέθη διαχωρισμών, το Union CSI δεν έχει τα μέσα για να ελέγχει ή να επηρεάζει τον προγραμματισμό των κατώτερων όγκων. Ακόμα και αν το Union CSI μπορούσε να παρακολουθεί με επιτυχία τη χρήση του αποθηκευτικού χώρου, το Union CSI απλώς δημιουργεί μικρότερα PVCs για να χειριστεί το κατώτερο πρόσθετο, και το API του PersistentVolumeClaim δεν είναι σε θέση να καθορίζει συγκεκριμένους δίσκους ή κόμβους στόχους.

Επιπλέον, σκεφτείτε το σενάριο, όπου το κατώτερο πρόσθετο χρονοδρομολογεί και εκχωρεί τους δύο κατώτερους τόμους στον ίδιο δίσκο, ο οποίος έχει αρκετό διαθέσιμο χώρο για και τους δύο. Το Union CSI δεν μπορεί να προβλέψει ή να αποτρέψει αυτό το σενάριο. Σε αυτήν την περίπτωση, ένας μόνο τόμος της αρχικής χωρητικότητας ήταν πιθανά εφικτός από το κατώτερο πρόσθετο εξαρχής, και τίποτα δεν επιτεύχθηκε με τη χρήση του Union CSI.

Σε αυτό το στάδιο, ο μόνος τρόπος για έναν χρήστη να επηρεάσει έμμεσα την χρονοδρομολόγηση και να αποκομίσει μία πρακτική περίπτωση χρήσης από το Union CSI, είναι να ζητήσει εσκεμμένα έναν τόμο αρκετά μεγάλο, ώστε να μην μπορεί να τοποθετηθεί σε έναν μόνο δίσκο στη συστοιχία, αλλά δύο τόμοι του μισού μεγέθους μπορούν να τοποθετηθούν σε διαφορετικούς δίσκους (και πιθανόν διαφορετικούς κόμβους), και αυτοί οι τόμοι μπορούν να προσπελαστούν από οπουδήποτε στη συστοιχία.

Ενώ η σχεδίαση αυτή του Union CSI είναι στατική και άκαμπτη, μας επιτρέπει να προχωρήσουμε προς τους στόχους μας χωρίς να περιορίζει τις προοπτικές για μία ισχυρή περίπτωση χρήσης. Συνεπώς, τα αποτελέσματά της θα μας βοηθήσουν να επιβεβαιώσουμε τις υποθέσεις μας και να επανέλθουμε με μια πιο εκλεπτυσμένη προσέγγιση σε μελλοντικές εκδόσεις.

2.5.3 Χώρος Ονομάτων

Κάθε στοιχείο και πόρος του Union CSI που αποτελεί μέρος της ανάπτυξης του στο Kubernetes, ή δημιουργείται άμεσα από αυτό κατά την εκτέλεσή του και ορίζεται εντός χώρου ονομάτων, θα δημιουργείται στον αφιερωμένο χώρο ονομάτων του προσθέτου. Ο χώρος ονομάτων του Union CSI ονομάζεται `union`.

2.5.4 StorageClass

Το Union CSI StorageClass αυτή τη στιγμή υποστηρίζει τις ακόλουθες παραμέτρους:

Παράμετρος	Τιμές	Προεπιλογή	Περιγραφή
`lowerStorageClassName`			Το όνομα του StorageClass του κατώτερου πρόσθετου τόμων. Η τιμή αυτή χρησιμοποιείται στο πεδίο `spec.storageClassName` των κατώτερων PVCs που δημιουργούνται από το Union CSI. Εάν η παράμετρος λείπει, τότε το πεδίο `spec.storageClassName` παραλείπεται από το πρόσθετο, και τα κατώτερα PVCs χρησιμοποιούν το προεπιλεγμένο (default) StorageClass της συστοιχίας.

Πίνακας 2.2: *Union CSI StorageClass parameters*

Οι παράμετροι του StorageClass περνούν στο χάρτη `CreateVolumeRequest.parameters`. Αυτή τη στιγμή, η μοναδική παράμετρος του StorageClass είναι το `lowerStorageClassName`, που καθορίζει το όνομα του κατώτερου StorageClass, που θα χρησιμοποιηθεί για τα αντικείμενα PersistentVolumeClaim, που δημιουργούνται από το Union CSI. Προτού οι χρήστες δημιουργήσουν PVCs με αυτό το StorageClass, οι διαχειριστές της συστοιχίας πρέπει να διασφαλίσουν ότι ένα κατώτερο StorageClass με το ίδιο όνομα έχει ρυθμιστεί και ένα αντίστοιχο κατώτερο πρόσθετο τόμων είναι εγκατεστημένο και λειτουργικό εντός της συστοιχίας.

Το StorageClass του πρόσθετου Union CSI χρησιμοποιεί τον τρόπο δέσμευσης τόμων `Immediate`, έτσι ώστε η δυναμική παροχή να ξεκινά αμέσως μόλις δημιουργηθεί ένα PersistentVolumeClaim από τον χρήστη. Η λειτουργία `Immediate` προτιμάται επειδή το Union CSI δεν έχει γνώση περί τοπολογίας προς το παρόν, οπότε δεν υπάρχει λόγος να καθυστερήσει η παροχή μέχρι να γίνει η χρονοδρομολόγηση ενός Pod σε ένα Node. Είναι σημαντικό να σημειωθεί ότι την πραγματική παροχή χειρίζεται το κατώτερο πρόσθετο τόμων, και το StorageClass του μπορεί να έχει ρυθμιστεί είτε σε `Immediate` είτε

σε `WaitForFirstConsumer`.

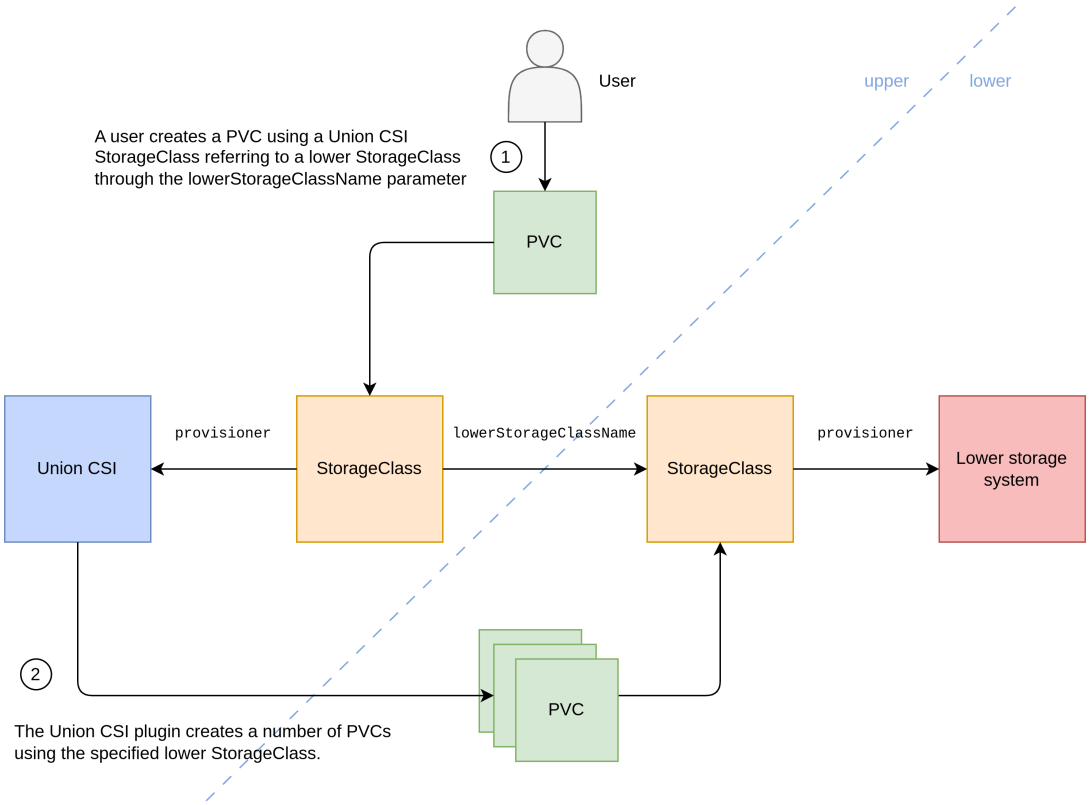
Η πολιτική ανάκτησης του StorageClass έχει οριστεί σε `Delete`, επιτρέποντας την αφαίρεση ενός άνω PersistentVolume όταν ο χρήστης διαγράφει το άνω PVC που είναι δεσμευμένο με το PV.

Η Λίστα 2.1 δείχνει ένα Union CSI StorageClass που καθορίζει ένα κατώτερο StorageClass με όνομα `lower-storageclass`.

Listing 2.1: *Παράδειγμα ενός Union CSI StorageClass*

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: union-storage
provisioner: union.csi.union.io
parameters:
  lowerStorageClassName: lower-storageclass
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Η Εικόνα 2.3 απεικονίζει τη σχέση μεταξύ του προσθέτου Union CSI και του κατώτερου συστήματος αποθήκευσης μέσω των αντίστοιχων αντικειμένων StorageClass.



Σχήμα 2.3: *Το Union CSI StorageClass και η σχέση του με το κατώτερο σύστημα αποθήκευσης*

2.5.5 Έννοια Τόμου

Οι οδηγοί CSI επικοινωνούν με ένα backend αποθήκευσης για να προωθήσουν τα αιτήματα του CO σχετικά με λειτουργίες τόμων και ερωτήματα. Αυτό το backend αποθήκευσης υλοποιείται από τον πάροχο αποθήκευσης και διαθέτει ένα API για την πρόσβαση στο προσφερόμενο σύστημα αποθήκευσης και υπηρεσίες. Επιπλέον, είναι υπεύθυνο για την καταγραφή και παρακολούθηση των δημιουργούμενων τόμων, των χαρακτηριστικών τους, της κατάστασης ζωής και υγείας τους και άλλες σχετικές πληροφορίες.

Όταν ένας CO ζητά τη δημιουργία ενός τόμου μέσω του CSI, παρέχει ένα μοναδικό αναγνωριστικό στο πεδίο `name` του `CreateVolumeRequest` του `CreateVolume` RPC. Τα πρόσθετα CSI μπορούν να επιλέξουν να χρησιμοποιήσουν αυτό το αναγνωριστικό απευθείας ή να παράξουν ένα νέο που να είναι κατάλληλο για χρήση με το API τους και για την αποθήκευση και χρήση του για την αναγνώριση του τόμου στο μέλλον.

Το Union CSI, εκμεταλλευόμενο το MergerFS, παρέχει τόμους βασισμένους στο FUSE σύστημα αρχείων που συνδυάζουν τα συστήματα αρχείων από το κατώτερο σύστημα αποθήκευσης. Επομένως, το Union CSI χρειάζεται έναν τρόπο, δεδομένου ενός μοναδικού αναγνωριστικού, περασμένου μέσω CSI RPCs, όπως τα `CreateVolume` και `ControllerPublishVolume`, για να αποφαίνεται εύκολα αν υπάρχει ένας αντίστοιχος Union CSI τόμος στην συστοιχία, να αποκτά γρήγορα πρόσβαση σε επιπλέον πληροφορίες σχετικά με αυτόν και να διατηρεί κατάσταση. Για να το επιτύχει αυτό, το Union CSI απαιτεί το δικό του backend αποθήκευσης.

Λαμβάνοντας υπόψη την παραδοχή μας για το Kubernetes ως τον CO για τον οποίο προορίζεται το Union CSI, το επεκτάσιμο API και τη βολική key-value αποθήκη etcd του, το να προσφύγουμε και να βασιστούμε στο API του Kubernetes και σε CustomResourceDefinitions (CRDs) ήταν μια εύκολη επιλογή.

Τα CRDs χρησιμοποιούνται συχνά στις εφαρμογές του Kubernetes. Οι προγραμματιστές μπορούν να συγχωνεύσουν το δηλωτικό τους API με αυτό του Kubernetes, να περιγράψουν τα σενάρια χρήσης και τη συμπεριφορά τους μέσω CRDs και να σχεδιάσουν ειδικούς χειριστές για τη διαχείριση των ειδικών αυτών πόρων. Επιπλέον, το Kubernetes προσφέρει ένα πλούσιο οικοσύστημα βιβλιοθηκών και εργαλείων που διευκολύνουν τη γρήγορη ανάπτυξη και δημοσίευση του API.

Επομένως, σχεδιάσαμε και αναπτύξαμε έναν ειδικό πόρο με το όνομα VolumeSplit. Η χρήση του API VolumeSplit έλυσε πολλές από τις δυσκολίες που προέκυψαν κατά την ανάπτυξη αυτού του πρόσθετου τόμων CSI, συμπεριλαμβανομένων:

- **Μοναδική πηγή αλήθειας**: Η ύπαρξη ενός αντικειμένου VolumeSplit υποδηλώνει την ύπαρξη ενός Union CSI τόμου.
- **Αποθήκευση και ανάκτηση δεδομένων**: Το πεδίο `spec` του VolumeSplit περιλαμβάνει πεδία παρόμοια με αυτά που περιέχονται στο `CreateVolumeRequest`. Όταν γίνεται μια κλήση `CreateVolume` στο πρόσθετο Union CSI, ένα αντικείμενο VolumeSplit συμπληρώνεται με τιμές από το `CreateVolumeRequest` και δημιουργείται στον διακομιστή API του Kubernetes. Οι επόμενες κλήσεις `CreateVolume` για τον ίδιο τόμο έχουν ως συνέπεια την ανάκτηση του αντίστοιχου αντικειμένου VolumeSplit από την αποθήκη key-value του Kubernetes. Στη συνέχεια, οι παράμετροι του αντικειμένου VolumeSplit συγκρίνονται με αυτές του `CreateVolumeRequest`, για να διασφαλιστεί η ταυτοδυναμία (idempotency).
- **Σύνολα από PersistentVolumeClaims**: Το Union CSI δημιουργεί μικρότερα PVCs ως απάντηση στο PVC ενός χρήστη. Ένα αντικείμενο VolumeSplit καθορίζει τον επιθυμητό αριθμό από PVCs με ίδιες προδιαγραφές. Ένας ειδικός χειριστής εποπτεύει τα VolumeSplits και τα συσχετιζόμενα αντίγραφα των PVCs στο παρασκήνιο, εκ μέρους του πρόσθετου Union CSI.

Το API VolumeSplit

Ο νέος ειδικός πόρος VolumeSplit, ορισμένος σε Go, φαίνεται στην Εικόνα 2.2:

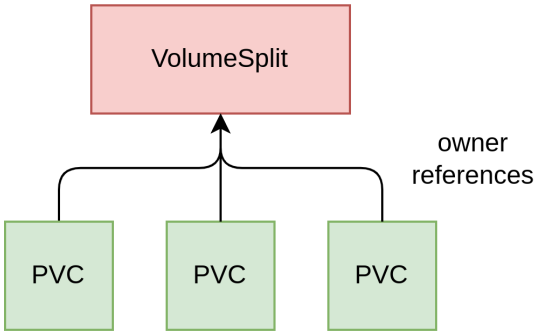
Listing 2.2: *Ο πόρος API VolumeSplit*

```go
// VolumeSplit is the Schema for the volumesplits API
type VolumeSplit struct {
    metav1.TypeMeta
    metav1.ObjectMeta

    Spec   VolumeSplitSpec
    Status VolumeSplitStatus
}

// VolumeSplitSpec defines the desired state of VolumeSplit
type VolumeSplitSpec struct {
    // Capacity is the desired minimum amount of storage size.
    // +optional
    Capacity *resource.Quantity

    // Number of desired PersistentVolumeClaims.
    // This is a pointer to distinguish between explicit zero and not specified.
    // Defaults to 1.
    // +optional
    Replicas *int32

    // Template describes the PersistentVolumeClaims that will be created.
    Template corev1.PersistentVolumeClaimTemplate
```

```
}

// VolumeSplitStatus defines the observed state of VolumeSplit
type VolumeSplitStatus struct {
    // AccessModes contains the actual access modes of the underlying volumes
    // backing the PersistentVolumeClaims managed by this VolumeSplit.
    // It is the intersection of status.accessModes for all the PersistentVolumeClaims.
    AccessModes []corev1.PersistentVolumeAccessMode

    // Capacity is the actual total capacity of the underlying volumes
    // backing the PersistentVolumeClaims managed by this VolumeSplit.
    // It is the sum of status.capacity.storage for all the PersistentVolumeClaims.
    // If nil it means that every status.capacity is nil.
    // +optional
    Capacity *resource.Quantity

    // Total number of PersistentVolumeClaims managed by this VolumeSplit.
    Replicas int32

    // Total number of PersistentVolumeClaims managed by this VolumeSplit that have "Bound" phase.
    BoundReplicas int32

    // Conditions contain the latest available observations of a VolumeSplit's current state.
    Conditions []VolumeSplitCondition
}

// VolumeSplitCondition describes the state of a VolumeSplit at a certain point.
type VolumeSplitCondition struct {
    // Type of VolumeSplit condition
    Type VolumeSplitConditionType
    // Status of the condition, one of True, False, Unknown.
    Status corev1.ConditionStatus
    // The last time the condition transitioned from one status to another.
    // +optional
    LastTransitionTime metav1.Time
    // The reason for the condition's last transition.
    // +optional
    Reason string
    // A human readable message indicating details about the transition.
    // +optional
    Message string
}

// VolumeSplitConditionType is a condition of a VolumeSplit.
type VolumeSplitConditionType string

const (
    // Ready means that:
    //
    // The actual number of replicas is equal to the desired number.
    // All replicas have "Bound" phase, i.e. each PVC is bound to a PV.
    // If spec.capacity is non-nil, status.capacity is greater than or equal to spec.capacity.
    // status.accessModes array is non-empty.
    VolumeSplitReady VolumeSplitConditionType = "Ready"
    // Pending means that the actual number of replicas is equal to the desired number
    // and at least one replica has "Pending" phase, i.e. at least one PVC is not bound to a PV.
    VolumeSplitPending VolumeSplitConditionType = "Pending"
    // Progressing means that the actual number of replicas has yet to meet the desired number.
    VolumeSplitReplicaProgressing VolumeSplitConditionType = "ReplicaProgressing"
    // ReplicaFailure means that a replica failed to be created or deleted.
    VolumeSplitReplicaFailure VolumeSplitConditionType = "ReplicaFailure"
)
```

Το API VolumeSplit ορίζει ρέπλικα από PVCs, με τρόπο παρόμοιο με αυτόν που ένα ReplicaSet ορίζει ρέπλικα από Pods. Το πεδίο `replicas` ορίζει τον επιθυμητό αριθμό των PVCs, ενώ το πεδίο `template` παρέχει τα πεδία `metadata` και `spec` για τα δημιουργηθέντα PVCs. Αξίζει να σημειωθεί ότι, στην παρούσα μορφή του, το VolumeSplit δεν περιλαμβάνει επιλογέα ετικετών (label selector) και δεν αποκτά υπάρχοντα PVCs μέσω ενός τέτοιου.

Η Εικόνα 2.3 απεικονίζει ένα παράδειγμα ενός VolumeSplit, που ορίζει τρία ρέπλι-



Σχήμα 2.4: *Ο ειδικός πόρος VolumeSplit και τα ρέπλικα PVCs του*

κα PVCs των 50GiB το καθένα και που ανήκουν στο StorageClass με όνομα `lower-storageclass`.

Listing 2.3: *Παράδειγμα ενός VolumeSplit*

```yaml
apiVersion: union.io/v1alpha1
kind: VolumeSplit
metadata:
  name: example-volumesplit
  namespace: union
spec:
  capacity: 150Gi
  replicas: 3
  template:
    spec:
      accessModes:
      - ReadWriteOnce
      storageClassName: lower-storageclass
      resources:
        requests:
          storage: 50Gi
```

Τα αντικείμενα VolumeSplit προορίζονται να δημιουργούνται από το πρόσθετο Union CSI στον χώρο ονομάτων `union` ως μέρος του RPC `CreateVolume`. Πώς ένα αντικείμενο VolumeSplit κατασκευάζεται από ένα αίτημα `CreateVolume`, φαίνεται στον Αλ-

γόριθμο 1.

Algorithm 1: Πως ένα VolumeSplit κατασκευάζεται από ένα `CreateVolumeRequest`

`name` ← ”`split-<CreateVolumeRequest.name>`”

`spec.replicas` ← 2

`spec.capacity` ←

`CreateVolumeRequest.capacity_range.required_bytes`

`spec.template.spec.accessModes` ← `[ReadWriteOnce]`

`spec.template.spec.resources.requests.storage` ←

`CreateVolumeRequest.capacity_range.required_bytes`/2

if ”`lowerStargeClassName`” *key exists in*

`CreateVolumeRequest.parameters` **then**

  `spec.template.spec.storageClassName` ←

  `CreateVolumeRequest.parameters[”lowerStorageClassName”]`

else

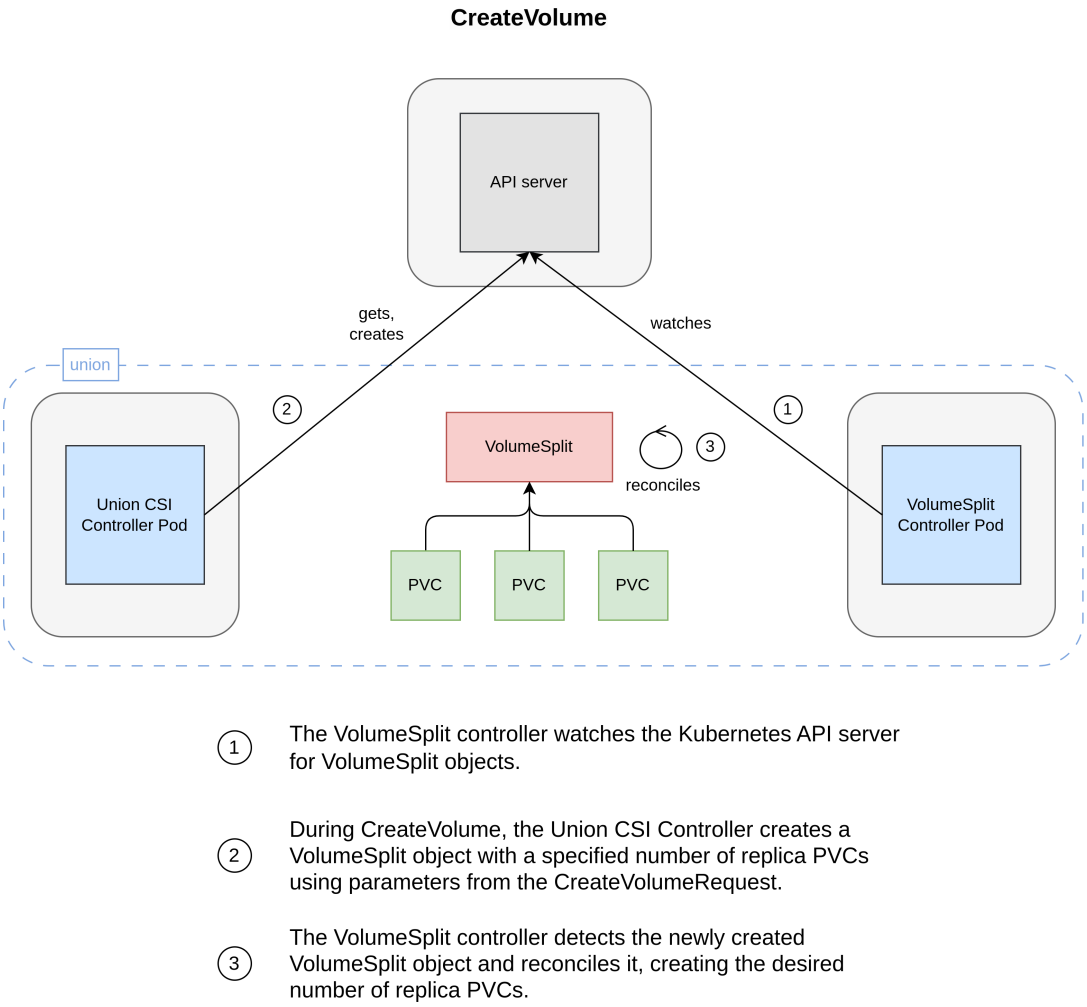
  `spec.template.spec.storageClassName` ← `nil` (the replica PVCs will use the default StorageClass of the cluster)

Ο χειριστής VolumeSplit διαχειρίζεται τα αντικείμενα VolumeSplit και τα ρέπλικα PVCs τους. Ο χειριστής δημιουργεί τον απαιτούμενο αριθμό PVCs στον ίδιο χώρο ονομάτων με το αντικείμενο VolumeSplit. Δεδομένου ότι το Union CSI δημιουργεί πάντα αντικείμενα VolumeSplit στον χώρο ονομάτων `union`, τα αντικείμενα PVCs στην πραγματικότητα θα βρίσκονται επίσης στον χώρο ονομάτων union. Παρατηρήστε, ότι το PVC του χρήστη (άνω) μπορεί να ορίζεται σε οποιοδήποτε χώρο ονομάτων έχει επιλεχθεί από τον χρήστη, αλλά τα κατώτερα PVCs που δημιουργούνται από το Union CSI, για να διαμερίσουν την αρχική αίτηση, βρίσκονται στον χώρο ονομάτων του πρόσθετου. Πώς αυτός ο αποθηκευτικός χώρος διαφορετικών χώρων ονομάτων γίνεται τελικά διαθέσιμος στον τελικό χρήστη, εξηγείται κατανοητά στην Ενότητα 2.5.6.

Ο Χειριστής VolumeSplit

Ο χειριστής VolumeSplit είναι υπεύθυνος για τη δημιουργία του επιθυμητού αριθμού PVCs που ορίζεται στο πεδίο `replicas` ενός αντικειμένου VolumeSplit, χρησιμοποιώντας το πρότυπο PVC από το πεδίο `spec.template`. Τα PVCs δημιουργούνται, με το πεδίο `metadata.ownerReferences` να έχει οριστεί ώστε να δείχνει στο αντικείμενο-

ιδιοκτήτη VolumeSplit και με το πεδίο `blockOwnerDeletion` να έχει αληθή τιμή, έτσι ώστε η διαγραφή του ιδιοκτήτη VolumeSplit να αποτρέπεται, μέχρι να διαγραφούν τα PVCs που ανήκουν σε αυτό, σε περίπτωση διαγραφής με αλληλουχία foreground.



Σχήμα 2.5: *Πως ένα αντικείμενο VolumeSplit και τα ρέπλικα PVCs του δημιουργούνται*

Επιπλέον, ο χειριστής παρατηρεί την κατάσταση του VolumeSplit και των αντίστοιχων PVCs που ανήκουν σε αυτό και ενημερώνει το πεδίο `status` του VolumeSplit. Ο χειριστής παράγει πληροφορίες σύνοψης σχετικά με την κατάσταση των κατώτερων τόμων, εξετάζοντας τα πεδία `status` των αντίστοιχων PVCs, και μεταφέρει αυτές τις πληροφορίες εντός του πίνακα `conditions` του VolumeSplit. Τα αντικείμενα PersistentVolumes των κατώτερων τόμων βρίσκονται εκτός του πεδίου ελέγχου του χειριστή και δεν παρακολουθούνται ούτε αναφέρονται από τον χειριστή.

Ο βρόχος συμφωνίας του χειριστή VolumeSplit, όσον αφορά τη διαχείριση των ρέπλικα

PVCs, είναι απλός:

1. Ο χειριστής καταλογίζει όλα τα PVCs στον ίδιο χώρο ονομάτων με το αντικείμενο VolumeSplit, που έχουν μια αναφορά ιδιοκτησίας στο VolumeSplit.
2. Αν τα PVCs που καταλογίστηκαν είναι λιγότερα από τα `spec.replicas`:
   (α′) Ο χειριστής δημιουργεί τον απαιτούμενο αριθμό των PVCs στον διακομιστή API, χρησιμοποιώντας το πεδίο `spec.template` του VolumeSplit και ορίζοντας το αντικείμενο VolumeSplit ως τον ιδιοκτήτη των PVCs, με το πεδίο `blockOwnerDeletion` να είναι αληθές.
3. Αν τα PVCs που καταλογίστηκαν είναι περισσότερα από τα `spec.replicas`:
   (α′) Ο χειριστής δεν κάνει τίποτα και απλώς ενημερώνει ότι η λειτουργία κλιμάκωσης προς τα κάτω δεν είναι υλοποιημένη. Η επιλογή, ποιά από τα πλεονάζοντα PVCs πρέπει να διαγραφούν, είναι μια δύσκολη εργασία και, σε περίπτωση που όλα τα PVCs είναι δεσμευμένα, μπορεί να οδηγήσει σε απώλεια δεδομένων. Επιπλέον, το πρόσθετο Union CSI δεν χρειάζεται αυτήν τη στιγμή λειτουργία μείωσης των PVCs.

Τα αντικείμενα VolumeSplits προορίζονται προς διαγραφή από το πρόσθετο Union CSI, ως μέρος του RPC `DeleteVolume`. Η διαγραφή με αλληλουχία foreground προτιμάται για την αποτροπή της απομάκρυνσης του αντικειμένου VolumeSplit, έως ότου αφαιρεθούν πρώτα όλα τα PVCs που του ανήκουν. Πρέπει να επισημάνουμε ότι, ακόμα και αν τα κατεχόμενα PVCs αφαιρούνται πριν, μετά ή ανεξάρτητα από το VolumeSplit, που καθορίζεται από τους τύπους διαγραφής με αλληλουχία foreground, background και orphan αντίστοιχα, τα αντίστοιχα PVs δεν συμπεριλαμβάνονται στην διαδικασία διαγραφής. Τα PVs μπορούν να αφαιρεθούν μόνο όταν αφαιρούνται οι υποκείμενοι τόμοι τους. Αυτή η διαδικασία μπορεί να διαρκέσει αρκετή ώρα, ακόμη και για πάντα, αν το αντίστοιχο πρόσθετο αποθήκευσης αντιμετωπίζει εσωτερικά σφάλματα ή έχει αφαιρεθεί από τη συστοιχία. Ο χειριστής VolumeSplit ενδιαφέρεται μόνο για το αντικείμενο VolumeSplit και τα PVCs που του ανήκουν, και τα αφαιρεί χρησιμοποιώντας αναφορές ιδιοκτησίας.

Ο χειριστής VolumeSplit αποτελεί ξεχωριστό δυαδικό αρχείο από αυτό του πρόσθετου Union CSI και προορίζεται για εγκατάσταση σε ξεχωριστό container στα πλαίσια ενός ανεξάρτητου Deployment στον χώρο ονομάτων `union`.

2.5.6 Έννοια Σύνδεσης Τόμου

Ακολουθώντας το ίδιο πνεύμα με την Ενότητα 2.5.5, το Union CSI πρέπει να ορίσει τι αποτελεί τη σύνδεση και αποσύνδεση των τόμων του, καθώς και πώς αυτές οι ενέργειες εκκινούνται, παρακολουθούνται και επαληθεύονται.

Ένας τόμος του Union CSI αποτελείται από έναν αριθμό κατώτερων τόμων ή *κλάδων*, οι οποίοι ζητούνται από το Union CSI μέσω των PersistentVolumeClaims στο κατώτερο πρόσθετο τόμων. Για να γίνει ένας τόμος του Union CSI προσβάσιμος σε έναν κόμβο, κάθε κλαδί πρέπει να είναι προσβάσιμο στον ίδιο κόμβο, είτε τοπικά, είτε απομακρυσμένα μέσω του δικτύου. Από αυτό το σημείο, οι κλάδοι πρέπει να συγχωνευτούν από μία διεργασία MergerFS στον κόμβο οικοδεσπότη, και το σύστημα αρχείων union πρέπει να προσαρτηθεί στο σύστημα αρχείων του οικοδεσπότη στον ειδικό χώρο του *Union CSI Node* (`/var/lib/union-csi/`).

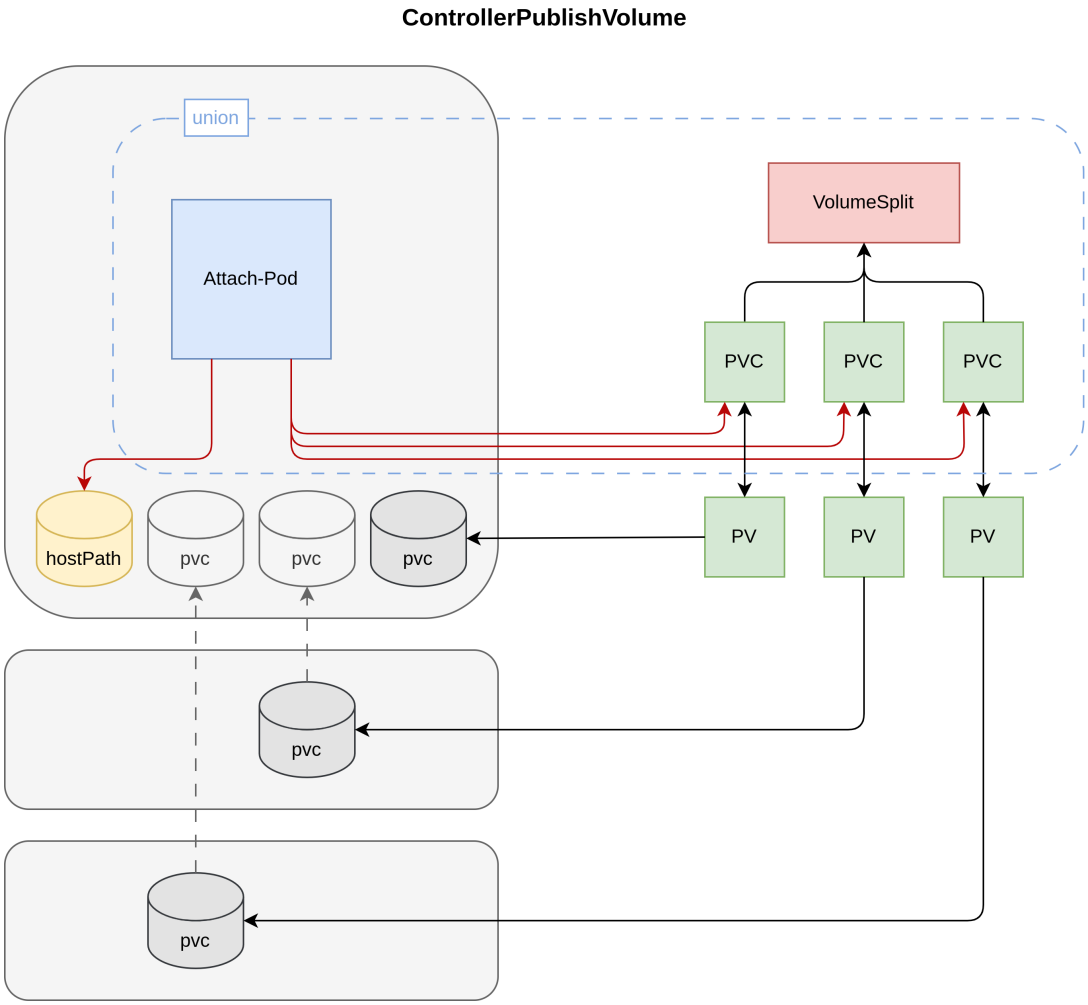
Για να αποφευχθεί η απευθείας εγκατάσταση του MergerFS σε κάθε κόμβο-οικοδεσπότη και για την βολική διαχείριση των κλάδων μέσω των PVCs τους, ένα ειδικό προνομιούχο Pod χειρίζεται τη διαδικασία σύνδεσης που περιγράφηκε παραπάνω. Αυτό το Pod χρησιμοποιεί τα PVCs των κλάδων του τόμου του Union CSI σε ένα container, που ενθυλακώνει το πρόγραμμα MergerFS, και ανατίθεται στον ίδιο κόμβο όπου θα εκτελεστεί η εργασία του χρήστη. Το πρόγραμμα MergerFS εκτελείται εντός του container, και το συγχωνευμένο σύστημα αρχείων εκδίδεται στον οικοδεσπότη. Αυτό το σύστημα αρχείων MergerFS στον κόμβο οικοδεσπότη αποτελεί τον πόρο αποθήκευσης ενός Union CSI τόμου, που αντιπροσωπεύεται στη συστοιχία από ένα αντικείμενο VolumeSplit.

Pod (ή Attach-Pod...)

Κατά τη διάρκεια του `ControllerPublishVolume`, το *Union CSI Controller* πρέπει να προσαρτήσει τον καθορισμένο τόμο στον καθορισμένο κόμβο. Το *Union CSI Controller* ανακτά το αντικείμενο VolumeSplit που αντιστοιχεί στο αναγνωριστικό τόμου του `ControllerPublishVolumeRequest`. Στη συνέχεια, δημιουργεί ένα Pod που χρησιμοποιεί ρέπλικα PVCs του VolumeSplit συν έναν επιπλέον τόμο `hostPath`, για να προσαρτήσει το σύστημα αρχείων στον κόμβο. Τα συστήματα αρχείων αυτά είναι όλα προσαρτημένα εντός του container MergerFS, που τρέχει σε προνομιούχα κατάσταση. Το Pod δημιουργείται στο API του Kubernetes στον ίδιο χώρο ονομάτων με τα PVCs και ανατίθεται στον καθορισμένο κόμβο.

Δεδομένου ότι αυτό το Pod χρησιμοποιεί έναν ή περισσότερους τόμους του κατώτερου πρόσθετου τόμων και χρονοδρομολογείται σε έναν κόμβο, το Kubernetes ενεργοποιεί τις ενέργειες σύνδεσης και προσάρτησης αυτών των τόμων στον ίδιο κόμβο από το κατώτερο πρόσθετο εκ μέρους του Union CSI. Το Union CSI χρειάζεται μόνο να περιμένει να μεταβεί η φάση του Pod σε Running, για να εξασφαλίσει ότι οι κατώτεροι τόμοι έχουν δημοσιευτεί επιτυχώς στον κόμβο από τον κατώτερο πρόσθετο, και το σύστημα αρχείων MergerFS έχει προσαρτηθεί στον οικοδεσπότη από το container του Pod.

Από εδώ και στο εξής, αυτό το συγκεκριμένο Pod, που δημιουργείται από το πρόσθετο Union CSI, θα αναφέρεται συχνά ως *"Attach-Pod"*, καθώς η δημιουργία του εκκινεί τη σύνδεση (και προσάρτηση) των κατώτερων τόμων. Το *Attach-Pod* δεν είναι ένα κάποιος νέος, ειδικός πόρος, αλλά απλά ένας χρήσιμο όρος που μπορεί να βοηθήσει στη διάκριση μεταξύ των Pods που δημιουργούνται από το χρήστη.



Σχήμα 2.6: *Ένα Attach-Pod που χρησιμοποιεί τα PVs ενός VolumeSplit και ένα τόμο HostPath*

Η Λίστα 2.4 παρουσιάζει ένα παράδειγμα ενός *Attach-Pod* για έναν τόμο με `volume_id` `pod-pvc-7d6...` και κόμβο με `node_id` `kind-worker2`.

Listing 2.4: *Example of an Attach-Pod*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f
  namespace: union
spec:
  containers:
  - args:
    - -c
    - gogomergerfs mergerfs --branches=/volume/branches/\*
      --target=/volume/merged --block
    command:
    - /bin/sh
    image: docker.io/on2e/gogomergerfs:dev-mergerfs2.37.1
    imagePullPolicy: Always
    name: gogomergerfs
    securityContext:
      privileged: true
    volumeMounts:
    - mountPath: /volume/branches/branch0-split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-b22xk
      name: branch0
    - mountPath: /volume/branches/branch1-split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-h8rgj
      name: branch1
    - mountPath: /volume/merged
      mountPropagation: Bidirectional
      name: target
  nodeSelector:
    kubernetes.io/hostname: kind-worker2
  volumes:
  - name: branch0
    persistentVolumeClaim:
      claimName: split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-b22xk
  - name: branch1
    persistentVolumeClaim:
      claimName: split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-h8rgj
  - hostPath:
      path: /var/lib/union-csi/volumes/pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f/merged
      type: DirectoryOrCreate
    name: target
```

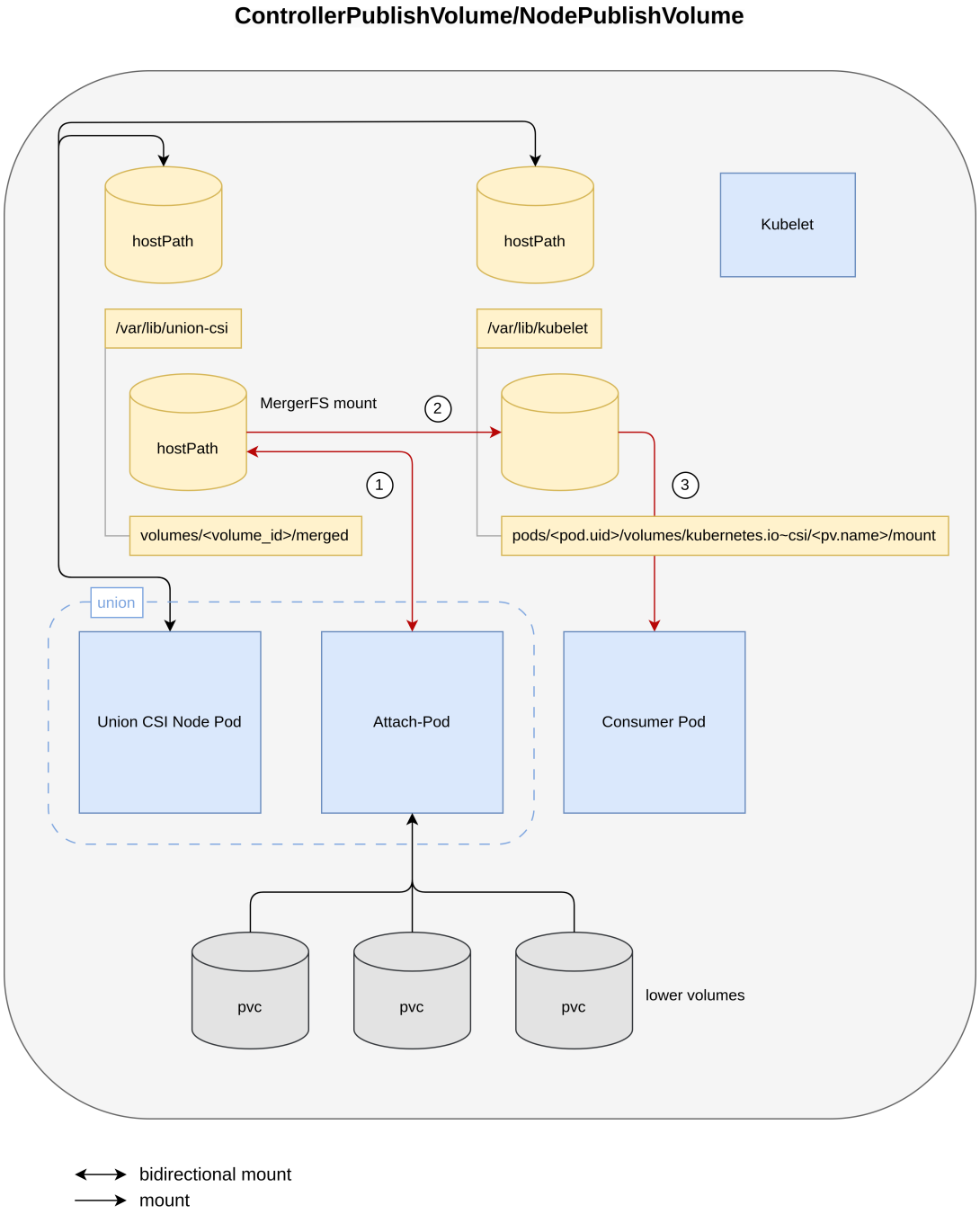
Το Pod της Λίστας 2.4 έχει όνομα `pod-<volume_id>` στον χώρο ονομάτων `union` και περιλαμβάνει:

- Το ακόλουθο container:
  - **`gogomergerfs`**: Το container εκτελεί την εντολή `mergerfs` μέσω ενός άλλου προγράμματος που ονομάζεται `gogomergerfs` και στη συνέχεια μπλοκάρει. Το container τρέχει σε προνομιούχα κατάσταση. Αυτό είναι απαραίτητο, επειδή το MergerFS χρειάζεται δικαιώματα χρήστη root και πρόσβαση στη συσκευή χαρακτήρων `/dev/fuse` στο σύστημα αρχείων του οικοδεσπότη. Μόλις γίνει η προσάρτηση του συστήματος αρχείων, το daemon πρόγραμμα FUSE του MergerFS τρέχει στον οικοδεσπότη, εκτελώντας τον ρόλο του διαμεσολαβητή μεταξύ των αιτημάτων συστήματος αρχείων του χρήστη και των υποκείμενων κλάδων. Για περισσότερες λεπτομέρειες σχετικά με το πρόγραμμα `gogomergerfs`, δείτε την επόμενη παράγραφο και την Ενότητα 3.2.1.

- Έναν επιλογέα κόμβων που χρησιμοποιεί την ακόλουθη ετικέτα:
  - **`kubernetes.io/hostname`**: Το Pod περιορίζεται ώστε να τρέξει μόνο στον κόμβο `<node_id>` με έναν επιλογέα κόμβων (node selector), που χρησιμοποιεί τη γνωστή ετικέτα `kubernetes.io/hostname`. Αυτή η ετικέτα, που έχει δεσμευτεί από το Kubernetes, τοποθετείται από το Kubelet σε αντικείμενα Node της συστοιχίας και περιέχει το όνομα του οικοδεσπότη υπολογιστή του κόμβου. Σημειώστε, ότι το όνομα του κόμβου μπορεί να αλλάξει σε σχέση με το "πραγματικό" όνομα του υπολογιστή με το πέρασμα της παραμέτρου `--hostname-override` στο `kubelet`. Ως αποτέλεσμα, αυτή η ετικέτα ενδέχεται να μην περιέχει πάντοτε το αναγνωριστικό του κόμβου όπως επιστρέφεται από την RPC `NodeGetInfo`. Μελλοντικές εκδόσεις του Union CSI μπορεί να χρησιμοποιήσουν μια διαφορετική, σταθερή ετικέτα, που ορίζεται από το ίδιο το Union CSI.
- Τους ακόλουθους τόμους:
  - **`branch*`**: Οι τόμοι τύπου `persistentVolumeClaim` των κατώτερων ρέπλικα PVCs προσαρτημένοι στο container εντός του καταλόγου `/volumes/-branches/`, ο καθένας στον δικό του υποκατάλογο. Αυτοί οι τόμοι αντιστοιχούν στους κλάδους του συστήματος αρχείων MergerFS.
  - **`target`**: Ο τόμος `hostPath` του μονοπατιού `/var/lib/union-csi/volumes/<volume_id>/merged` που είναι προσαρτημένος εντός του container στο μονοπάτι `/volumes/merged`. Αυτό είναι το σημείο προσάρτησης του συστήματος αρχείων MergerFS. Ο τόμος προσαρτάται με αμφίδρομη κατεύθυνση προσάρτησης, έτσι ώστε το προσαρτημένο σύστημα αρχείων να μεταδοθεί στον οικοδεσπότη κάτω από το `/var/lib/union-csi/` και το *Union CSI Node* να μπορεί να τον προσαρτήσει στο καθορισμένο μονοπάτι-στόχο κατά τη διάρκεια του `NodePublishVolume`.

Αφού ένα *Attach-Pod* επιτύχει τη συγχώνευση και την προσάρτηση του Union CSI τόμου (το σύστημα αρχείων MergerFS) στον κόμβο οικοδεσπότη, συνεχίζει να τρέχει στον κόμβο. Το *Union CSI Controller* ερμηνεύει τη τιμή `Running` του πεδίου `spec.phase` και την τιμή `<node_id>` του πεδίου `spec.nodeName` του Pod, ως επιβεβαίωση ότι ο Union CSI τόμος είναι συνδεδεμένος και προσαρτημένος στο καθορισμένο Node.

Στη συνέχεια, κατά τη διάρκεια του `ControllerUnpublishVolume`, το *Union CSI Controller*



1 During ControllerPublishVolume, the Attach-Pod mounts the MergerFS filesystem on the host under /var/lib/union-csi.

2 During NodePublishVolume, the Union CSI Node bind-mounts the MergerFS filesystem at the target path under /var/lib/kubelet.

3 Kubelet bind-mounts the target path into the consumer Pod container(s).

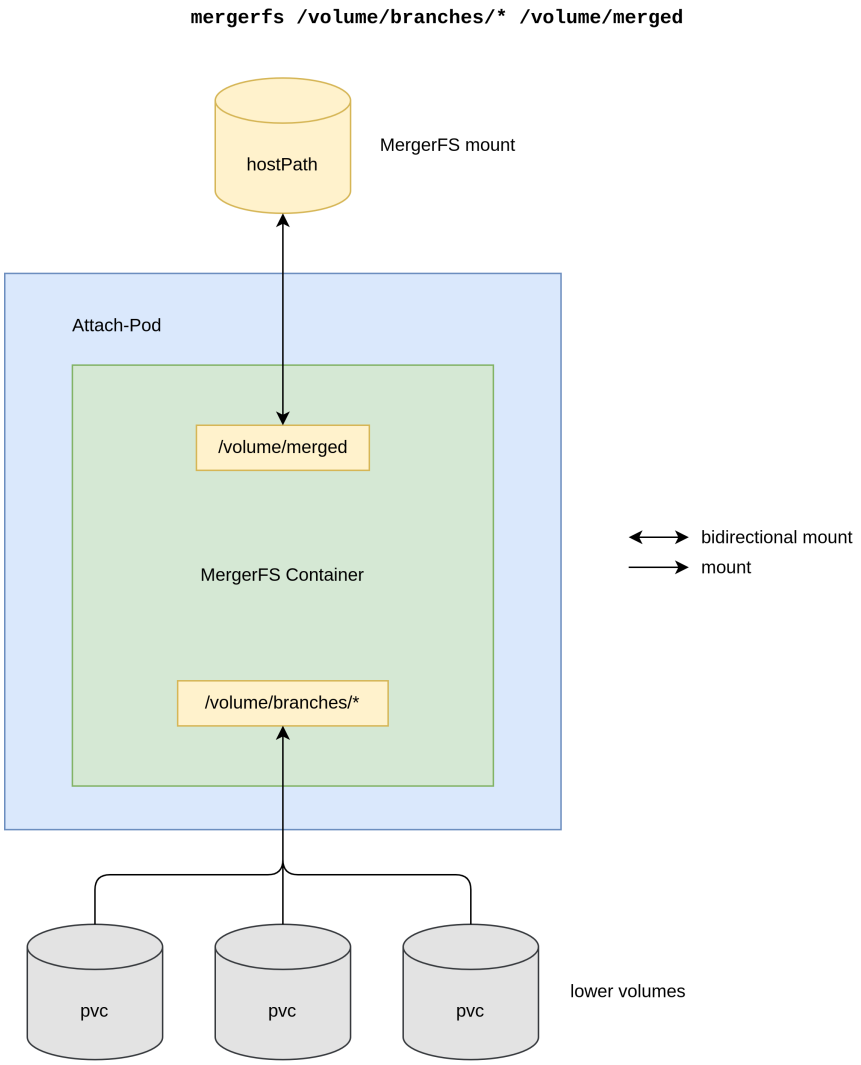
Σχήμα 2.7: *Πως ένας Union CSI τόμος προσαρτάται σε ένα Pod καταναλωτή*

ανακτά και διαγράφει το αντίστοιχο *Attach-Pod* και περιμένει την αφαίρεσή του από τον διακομιστή API, ως επιβεβαίωση ότι ο Union CSI τόμος είναι αποσυνδεδεμένος. Όταν το *Attach-Pod* διαγράφεται, το σύστημα αρχείων MergerFS αποπροσαρτάται ομαλά από τον κόμβο-οικοδεσπότη και οι τόμοι του κατώτερου συστήματος αποθήκευσης αποσυνδέονται από τον κόμβο. Όταν ένας χρήστης δημιουργεί ένα νέο Pod καταναλωτή, που χρησιμοποιεί το Union CSI τόμο, είτε στον ίδιο, είτε σε διαφορετικό κόμβο, το *Union CSI Controller* επανεκκινεί τη διαδικασία. Δημιουργεί ξανά ένα *Attach-Pod*, χρησιμοποιώντας τους κατώτερους τόμους για να τους συγχωνεύσει ξανά και να προσαρτήσει το Union CSI τόμο στον καθορισμένο κόμβο.

MergerFS container

Μέσα στο container MergerFS, κάθε ένας από τους τόμους PVC (είσοδοι) προσαρτάται κάτω από τον κατάλογο `/volume/branches/`, ενώ ο τόμος `hostPath` (έξοδος) προσαρτάται στον κατάλογο `/volume/merged`. Η παράμετρος ”branches” του `mergerfs` τίθεται σε `/volume/branches/*` για να συμπεριλάβει όλους τους καταλόγους κάτω από το `/volume/branches/`, και η παράμετρος ”mountpoint” τίθεται σε `/volume/merged`. Μετά την εκτέλεση του `mergerfs`, το προκύπτον σύστημα αρχείων προσαρτάται στο `/volume/merged` και διαδίδεται στον κόμβο οικοδεσπότη.

Μετά από μια επιτυχημένη συγχώνευση, το *Attach-Pod* πρέπει να παραμείνει ενεργό στον κόμβο, για να εξυπηρετεί το σύστημα αρχείων MergerFS μέχρι να διαγραφεί από το Union CSI κατά τη διάρκεια του `ControllerUnpublishVolume` RPC. Ωστόσο, η επιτυχής εκτέλεση του `mergerfs` προσαρτά το προκύπτον σύστημα αρχείων union και εξέρχεται; δεν μπλοκάρει. Η διεργασία `mergerfs` μετατρέπεται σε daemon: η τρέχουσα διαδικασία εκτελεί `fork` και η γονική διαδικασία σκοτώνεται. Η νέα διακλαδωμένη διαδικασία είναι ο χειριστής FUSE του συστήματος αρχείων, που τρέχει στο παρασκήνιο και επεξεργάζεται τα αιτήματα E/E του χρήστη. Αυτό σημαίνει ότι αν εκτελέσουμε ένα container με το `mergerfs` ως σημείο εισόδου, το `mergerfs` θα τρέξει μέσα στο container, θα εξέλθει με κατάσταση εξόδου 0 σε περίπτωση επιτυχίας, ο χειριστής FUSE και όλες οι διαδικασίες του container θα τερματίσουν και το container θα τερματίσει. Ένα Pod που τρέχει ένα τέτοιο container θα μεταβεί από την κατάσταση `Running` στην κατάσταση `Succeeded` και θα παραμείνει σε αυτή μέχρι να απομακρυνθεί. Το container MergerFS του *Attach-Pod* πρέπει να μπλοκάρει μετά από μια επιτυχημένη προσάρτηση, ώστε ο χειριστής FUSE του MergerFS να μπορεί να συνεχίσει να λειτουργεί.



Σχήμα 2.8: *Το MergerFS container ενός Attach-Pod και οι προσαρτημένοι τόμοι του*

Ένα άλλο πρόβλημα προκύπτει, όταν ένα container MergerFS τερματίζει, ενώ το σύστημα αρχείων MergerFS είναι προσαρτημένο στον οικοδεσπότη με αμφίδρομη κατεύθυνση προσάρτησης (`rshared` στο Linux). Εάν το container τερματίσει πριν από την αποσύνδεση του συστήματος αρχείων, το daemon FUSE τερματίζεται επίσης. Οποιεσδήποτε προσαρτήσεις του συστήματος αρχείων έξω από το container, τώρα θα φαίνονται ως κατεστραμμένες. Αυτό συμβαίνει επειδή η διαδικασία χειριστή, που είχε καταχωρηθεί με τον πυρήνα για να προωθήσει τα αιτήματα E/E του συστήματος αρχείων, δεν υπάρχει πλέον. Οι κατεστραμμένες προσαρτήσεις πρέπει να αποσυνδεθούν, πριν μπορέσουν να χρησιμοποιηθούν ξανά. Η Λίστα 2.5 δείχνει, πώς μια κατεστραμ-

μένη προσάρτηση MergerFS εμφανίζεται στο τερματικό μετά το σκότωμα του χειριστή MergerFS και πώς ο αρχικός κατάλογος αποκαθίσταται μετά την αποπροσάρτηση του συστήματος αρχείων.

Listing 2.5: *Το σύστημα αρχείων MergerFS εμφανίζεται κατεστραμμένο αν ο χειριστής FUSE σκοτωθεί*

```
# ls -il

total 12
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
6291862 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint

# mergerfs branch1:branch2 mountpoint
# ls -il

total 12
            6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
            6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
5424983562661234939 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint

# ps aux | grep mergerfs

root     1718664  0.0  0.0 901132  5076 ?        S<sl 23:01   0:00 mergerfs branch1:branch2 ↩
     mountpoint
root     1718815  0.0  0.0   9088  2432 pts/7    S+   23:01   0:00 grep --color=auto mergerfs

# kill -9 1718664
# ls -il

ls: cannot access 'mountpoint': Transport endpoint is not connected
total 8
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
      ? d????????? ? ?    ?       ?            ? mountpoint

# touch mountpoint/file
touch: cannot touch 'mountpoint/file': Transport endpoint is not connected

# umount mountpoint
# ls -il

total 12
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
6291862 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint
```

Οι τόμοι Union CSI είναι συστήματα αρχείων MergerFS που προσαρτώνται σε έναν κόμβο μέσω αμφίδρομης κατεύθυνσης προσαρτήσεων από containers MergerFS σε Pods. Εάν ένα container τερματιστεί χωρίς να αποσυνδέσει πρώτα το σύστημα αρχείων, ο κατάλογος οικοδεσπότη στο `/var/lib/union-csi/volumes/<volume_id>/merged` θα καταστραφεί. Σε τέτοια περίπτωση, ο κατάλογος θα πρέπει να αποπροσαρτηθεί χειροκίνητα από το εκάστοτε στιγμιότυπο του *Union CSI Node* στον κόμβο. Διαφορετικά, νέα Pods χρηστών που προσπαθούν να χρησιμοποιήσουν αυτό το Union CSI τόμο στον ίδιο κόμβο θα αποτύχουν, επειδή ο κατεστραμμένος κατάλογος `hostPath` δεν μπορεί πλέον να προσαρτηθεί από τον Kubelet σε ένα νέο *Attach-Pod*. Επομένως, είναι σημαντικό για το container Mergerfs του *Attach-Pod* να αποπροσαρτήσει το σύστημα αρχείων από τον κατάλογο προορισμού */volume/merged/* (και επομένως από τον αντίστοιχο κατάλογο στον οικοδεσπότη), όταν το Pod είναι σε διαδικασία τερματισμού.

Για να αντιμετωπιστούν αυτές οι απαιτήσεις, γράψαμε ένα πρόγραμμα σε Go, που πλαισιώνει την εντολή `mergerfs`. Το πρόγραμμα `gogomergerfs`, όπως φαίνεται στο παράδειγμα του Pod στη Λίστα 2.4, εκτελεί την εντολή `mergerfs`, περιμένει για σήματα UNIX (`SIGTERM` και `SIGINT`) και προχωρά στην αποπροσάρτηση του συστήματος αρχείων από τον κατάλογο προορισμού. Για λεπτομέρειες σχετικά με το πρόγραμμα επισκεφθείτε την Ενότητα 3.2.1.

2.5.7 Βοηθητικοί Containers

Όπως περιγράφηκε στην Ενότητα 2.5.1, ο οδηγός Union CSI υλοποιεί τα ακόλουθα RPCs, που περιλαμβάνουν λειτουργίες τόμων:

- **`CreateVolume`**
- **`DeleteVolume`**
- **`ControllerPublishVolume`**
- **`ControllerUnpublishVolume`**
- **`NodePublishVolume`**
- **`NodeUnpublishVolume`**

Συνεπώς, οι βοηθητικοί containers του Kubernetes, που απαιτούνται από και περιλαμβάνονται μαζί με τον οδηγό Union CSI, είναι:

- **`external-provisioner`** (καλεί τα RPCs `CreateVolume`/`DeleteVolume`)
- **`external-attacher`** (καλεί τα RPCs `ControllerPublishVolume`/`ControllerUnpublishVolume`)
- **`node-driver-registrar`** (καταχωρεί το πρόσθετο με το Kubelet το οποίο καλέι τα RPCs `NodePublishVolume`/`NodeUnpublishVolume`)

2.6 Παραδείγματα

Σε αυτήν την ενότητα, παρουσιάζουμε εν συντομία τη ροή των κύριων λειτουργιών του Union CSI, βασισμένη στην ανάλυση που παρουσιάστηκε στις προηγούμενες ενότητες και ακολουθώντας τη γενική ροή επιτυχίας.

2.6.1 Δημιουργία Τόμων

1. Ένας διαχειριστής συστοιχίας δημιουργεί ένα αντικείμενο StorageClass, που αναφέρεται στον οδηγό Union CSI και ένα αντικείμενο StorageClass, που αναφέρεται σε ένα κατώτερο πρόσθετο τόμων.
2. Ένας χρήστης δημιουργεί ένα PersistentVolumeClaim, στον χώρο ονομάτων της επιλογής του, που αναφέρεται στο StorageClass του Union CSI.
3. Το βοηθητικό container `external-provisioner` του οδηγού Union CSI ανιχνεύει το αδέσμευτο PVC και καλεί το RPC `CreateVolume` του οδηγού.
4. Εάν η κλήση `CreateVolume` επιστρέψει επιτυχώς, το `external-provisioner` δημιουργεί ένα αντικείμενο PersistentVolume για να αντιπροσωπεύει τον νέο τόμο και τον δένει με το PersistentVolumeClaim.

Το RPC `CreateVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο 2.

Algorithm 2: Το RPC `CreateVolume` του πρόσθετου Union CSI

request : `CreateVolumeRequest`

response: `CreateVolumeResponse, error`

1. **if** `CreateVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `CreateVolumeRequest.name`
   `capacity` ← `CreateVolumeRequest.capacity_range.required_bytes`
   `class` ← `CreateVolumeRequest.parameters["lowerStorageClassName"]`
   `rwo` ← `ReadWriteOnce`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit exists* **then**
   compare the VolumeSplit parameters with those from the `CreateVolumeRequest` (idempotency):
   if `spec.capacity` < `capacity` **or**
   `spec.template.spec.storageClassName` ≠ `class` **or**
   `spec.template.spec.accessModes` ≠ `[rwo]`
   then return with `ALREADY_EXISTS` error code and a related message.
   else
   create a new VolumeSplit `union/split-<volumeID>`. The VolumeSplit fields are populated using values from the `CreateVolumeRequest`, as described in Algorithm 8. The VolumeSplit controller creates the desired replica PVCs in the same namespace as the VolumeSplit.
5. Poll the VolumeSplit object using exponential backoff for a short period waiting for one of the following conditions:
   (α′) **if** *VolumeSplit* `status` *contains a condition type of* `Ready` *or* `Pending` *with condition status of* `True` **then** polling succeeds.
   (β′) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.
6. **if** *polling succeeds* **then**
   `CreateVolumeResponse.volume.volume_id` ← `volumeID`
   `error` ← `nil`

2.6.2 Διαγραφή Τόμων

1. Ένας χρήστης διαγράφει το PVC που είναι δεμένο με το PV, που υποστηρίζεται από τον Union CSI τόμο.
2. Το `external-provisioner` του οδηγού Union CSI ανιχνεύει τη διαγραφή του PVC και καλεί το RPC `DeleteVolume` του οδηγού.
3. Εάν η κλήση `DeleteVolume` επιστρέψει επιτυχώς, το `external-provisioner` διαγράφει το απελευθερωμένο αντικείμενο PV.

Το RPC `DeleteVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο 3.

Algorithm 3: Το RPC `DeleteVolume` του πρόσθετου Union CSI

request : `DeleteVolumeRequest`

response: `DeleteVolumeResponse, error`

1. **if** `DeleteVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `DeleteVolumeRequest.volume_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `OK` error code (idempotency) .
5. **if** `deletionTimestamp` $\neq$ `nil` **then** delete the VolumeSplit using foreground cascading deletion (so the owned PVCs are removed before the VolumeSplit) .
6. Poll the VolumeSplit object using exponential backoff for a short period waiting for one of the following conditions:

   (α′) **if** *VolumeSplit is removed from the API server* **then** polling succeeds.

   (β′) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

   The `DeleteVolume` request may time out during this period.
7. **if** *polling succeeds* **then**
   - `DeleteVolumeResponse` ← `{}`
   - `error` ← `nil`

2.6.3 Σύνδεση Τόμων

1. Ένας χρήστης δημιουργεί ένα Pod που χρησιμοποιεί το PVC, που αναφέρεται στον Union CSI τόμο.

2. Ο χρονοδρομολογητής επιλέγει ένα Node για το Pod και το εσωτερικό πρόσθετο τόμων CSI δημιουργεί ένα αντικείμενο VolumeAttachment.
3. Το βοηθητικό container `external-attacher` του οδηγού Union CSI ανιχνεύει το VolumeAttachment και καλεί το RPC `ControllerPublishVolume` του οδηγού.
4. Εάν η κλήση `ControllerPublishVolume` επιστρέψει επιτυχώς, το `external-attacher` ανανεώνει το αντικείμενο VolumeAttachment για να σημάνει την επιτυχή σύνδεση του τόμου.

Το RPC `ControllerPublishVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο 4.

Algorithm 4: Το RPC `ControllerPublishVolume` του πρόσθετου Union CSI

request : `ControllerPublishVolumeRequest`

response: `ControllerPublishVolumeResponse, error`

1. **if** `ControllerPublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `ControllerPublishVolumeRequest.volume_id`
   `nodeID` ← `ControllerPublishVolumeRequest.node_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
5. Get Node object `<nodeID>`.
6. **if** *Node does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
7. Get Pod object `union/pod-<volumeID>`.
8. **if** *Pod does NOT exist* **then** create a new Pod `union/pod-<volumeID>`. The Pod carries the MergerFS container, uses the PVCs of the VolumeSplit and a `hostPath` volume, and is assigned to Node `<nodeID>`, as described in Section 3.5.13. The Pod merges the PVC volumes and mounts the MergerFS filesystem on the `hostPath` volume at **"**`/var/lib/union-csi/volumes/<volumeID>/merged/`**"** on the host .
9. Poll the Pod using exponential backoff for a short period waiting for one of the following conditions:
   (α′) **if** *Pod is terminating (has phase of* `Failed` *or* `Succeeded`*)* **then return** with `INTERNAL` error code and a related message.
   (β′) **if** *Pod is running on a Node that is not* `<nodeID>` **then return** with `FAILED_PRECONDITION` error code and a related message.
   (γ′) **if** *Pod is running on Node* `<nodeID>` **then** polling succeeds.
   (δ′) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

   The `ControllerPublishVolume` request may time out during this period.
10. **if** *polling succeeds* **then**
    `ControllerPublishVolumeResponse` ← `{}`
    `error` ← `nil`

2.6.4 Αποσύνδεση Τόμων

1. Ο χρήστης διαγράφει το Pod που αναφέρεται στον Union CSI τόμο και έχει χρονοδρομολογηθεί σε ένα Node.
2. Το εσωτερικό πρόσθετο τόμων CSI διαγράφει το αντίστοιχο αντικείμενο VolumeAttachment που προστατεύεται από ένα finalizer.
3. Το `external-attacher` του οδηγού Union CSI ανιχνεύει το `deletionTimestamp` στο VolumeAttachment και καλεί το RPC `ControllerUnpublishVolume` του οδηγού.
4. Εάν η κλήση `ControllerUnpublishVolume` επιστρέψει επιτυχώς, το `external-attacher` αφαιρεί το finalizer από το VolumeAttachment, για να επιτραπεί η διαγραφή του από το API και να σημάνει την επιτυχή αποσύνδεση του τόμου.

Το RPC `ControllerUnpublishVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο

5.

Algorithm 5: Το RPC `ControllerUnpublishVolume` του πρόσθετου Union CSI

request : `ControllerUnpublishVolumeRequest`

response: `ControllerUnpublishVolumeResponse, error`

1. **if** `ControllerUnpublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `ControllerUnpublishVolumeRequest.volume_id`
   `nodeID` ← `ControllerUnpublishVolumeRequest.node_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
5. Get Node object `<nodeID>`.
6. **if** *Node does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
7. Get Pod object `union/pod-<volumeID>`.
8. **if** *Pod does NOT exist* **then return** with `OK` error code (idempotency).
9. **if** *Pod is scheduled on a Node that is not* `<nodeID>` *or is not yet scheduled* **then return** with `OK` error code (idempotency).
10. **if** *Pod is scheduled on Node* `<nodeID>` *and* `deletionTimestamp` $\neq$ `nil` **then** delete the Pod.
11. Poll the Pod using exponential backoff for a short period waiting for one of the following conditions:

    (α′) **if** *Pod is removed from the API server* **then** polling succeeds.

    (β′) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

    The `ControllerUnpublishVolume` request may time out during this period.
12. **if** *polling succeeds* **then**
    `ControllerUnpublishVolumeResponse` ← `{}`
    `error` ← `nil`

2.6.5 Προσάρτηση Τόμων

1. Το Kubelet ανιχνεύει ότι το Pod που αναφέρεται σε έναν Union CSI τόμο έχει χρονοδρομολογηθεί στο Node και καλεί το RPC `NodePublishVolume` του οδηγού Union CSI, μέσω του καταχωρημένου UNIX domain socket.
2. Εάν η κλήση `NodePublishVolume` επιστρέψει επιτυχώς, το καθορισμένο μονοπάτι προορισμού, όπου ο Union CSI τόμος είναι προσαρτημένος, προσαρτάται από το Kubelet εντός του container του Pod.

Το RPC `NodePublishVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο 6.

Algorithm 6: Το RPC `NodePublishVolume` του πρόσθετου Union CSI

request : `NodePublishVolumeRequest`

response: `NodePublishVolumeResponse, error`

1. **if** `NodePublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `NodePublishVolumeRequest.volume_id`
   `targetPath` ← `NodePublishVolumeRequest.target_path`
   `sourcePath` ← ”`/var/lib/union-csi/volumes/<volumeID>/merged/`”
3. **if** `<targetPath>` *does NOT exist on the host or is corrupted* **then return** with `INTERNAL` error code and a related message .
4. **if** `<sourcePath>` *does NOT exist on the host or is corrupted* **then return** with `INTERNAL` error code and a related message .
5. **if** `<sourcePath>` *is ALREADY mounted on* `<targetPath>` **then return** with `OK` error code (idempotency).
6. Mount (bind) `<sourcePath>` on `<targetPath>`.
7. **if** *mounting succeeds* **then**
   `NodePublishVolumeResponse` ← `{}`
   `error` ← `nil`

2.6.6 Αποπροσάρτηση Τόμων

1. Ο Kubelet ανιχνεύει ότι το Pod που αναφέρεται στον Union CSI τόμο τερματίστηκε ή διαγράφηκε στο Node και καλεί το RPC `NodeUnpublishVolume` του οδηγού Union CSI, μέσω του καταχωρημένου UNIX domain socket.

2. Εάν η κλήση `NodeUnpublishVolume` επιστρέψει επιτυχώς, το καθορισμένο μονοπάτι προορισμού, όπου ο Union CSI τόμος ήταν προσαρτημένος, αποπροσαρτάται από το container του Pod.

Το RPC `NodeUnpublishVolume` του Union CSI παρουσιάζεται στον Αλγόριθμο 7.

Algorithm 7: Το RPC `NodeUnpublishVolume` του πρόσθετου Union CSI

request : `NodeUnpublishVolumeRequest`

response: `NodeUnpublishVolumeResponse, error`

1. **if** `NodeUnpublishVolumeRequest` *is missing required fields or has invalid values* **then** **return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `NodeUnpublishVolumeRequest.volume_id`
   `targetPath` ← `NodeUnpublishVolumeRequest.target_path`
3. **if** `<targetPath>` *does NOT exist on the host or is corrupted* **then** **return** with `INTERNAL` error code and a related message .
4. **if** `<targetPath>` *is not mounted by a filesystem* **then return** with `OK` error code (idempotency).
5. Unmount `<targetPath>`.
6. **if** *unmounting succeeds* **then**
   - `NodeUnpublishVolumeResponse` ← `{}`
   - `error` ← `nil`

3 Υλοποίηση

Σε αυτό το κεφάλαιο, παρέχουμε μια συνοπτική περιγραφή των λεπτομερειών υλοποίησης της εφαρμογής Union CSI.

3.1 Επισκόπηση

Η υλοποίηση της εφαρμογής Union CSI αποτελείται από τρία διακριτά τμήματα:

- **Gogomergerfs:** Το πρόγραμμα που πλαισιώνει και εκτελεί την εντολή `mergerfs`.
- **VolumeSplit Controller:** Ο ειδικός πόρος VolumeSplit και ο χειριστής που τον διαχειρίζεται.
- **Union CSI:** Ο οδηγός CSI και το backend αποθήκευσης.

Κάθε τμήμα έχει γραφτεί στη γλώσσα προγραμματισμού Go και ο κώδικάς του βρίσκεται στο ίδιο αποθετήριο με όνομα ”Union CSI”, εντός του δικού του καταλόγου. Επιπλέον, κάθε τμήμα μεταγλωττίζεται σε ένα ξεχωριστό δυαδικό αρχείο και πακετάρεται σε εικόνα Docker, χρησιμοποιώντας τα δικά του Makefile και Dockerfile αρχεία.

Το container Gogomergerfs χρησιμοποιείται και αναπτύσσεται από το πρόσθετο Union CSI κάθε φορά που δημιουργεί ένα Pod (*Attach-Pod*) στο Kubernetes, για να ενώσει τους χαμηλότερους τόμους.

Τόσο ο χειριστής VolumeSplit, όσο και το Union CSI, αναπτύσσονται σε containers σε συστοιχία Kubernetes χρησιμοποιώντας αρχεία YAML. Αρχεία *Kustomization* (`kustomization.yaml`) διευκολύνουν την εγκατάσταση και απεγκατάσταση ολόκληρης της ε-

φαρμογής και των πόρων της στο Kubernetes με μία απλή εντολή `kubectl apply -k` ή `delete -k`.

Το Union CSI είναι συμβατό με:

- **Kubernetes:** v1.26+
- **CSI specification:** v1.8.0
- **MergerFS:** v2.37.0
- **OS:** Linux

3.2 Τμήματα

Σε αυτήν την ενότητα, εξετάζουμε την υλοποίηση των βασικών συστατικών της εφαρμογής Union CSI, προσφέροντας μία εικόνα της εσωτερικής τους αρχιτεκτονικής.

3.2.1 Gogemergerfs

Το Gogomergerfs είναι ένα πρόγραμμα ”wrapper” σε Go για την εντολή `mergerfs`. Σχεδιάστηκε να χρησιμοποιείται από το πρόσθετο Union CSI για τον συνδυασμό τόμων σε ένα σύστημα αρχείων MergerFS εντός ενός Pod και να προσφέρει συγκεκριμένες λειτουργίες για τη βελτίωση της εμπειρίας χρήστη, αφού το σύστημα αρχείων προσαρτηθεί.

Η κύρια υποεντολή του `gogomergerfs` είναι η `mergerfs`, η οποία εκτελεί τό πρόγραμμα `mergerfs`. Οι διαθέσιμες παράμετροι για αυτήν την υποεντολή φαίνονται στον Πίνακα 3.1:

Η εντολή `gogomergerfs mergerfs` προσπαθεί να μιμηθεί τη σύνταξη του αρχικού προγράμματος `mergerfs` στη γραμμή εντολών. Η εντολή

```
gogomergerfs mergerfs -o=<o1>,<o2> --branches=b1,b2,b3 --target=t
```

καλεί την

```
mergerfs -o <o1>,<o2> b1:b2:b3 t
```

διατηρώντας την ίδια συμπεριφορά σαν να εκτελείται το πρόγραμμα `mergerfs` απευθείας.

Παράμετρος	Τιμές	Προεπιλογή	Περιγραφή
`--branches`			Λίστα χωριζόμενη με κόμμα των μονοπατιών προς συγχώνευση.
`--target`			Το σημείο προσάρτησης του συστήματος αρχείων union.
`-o, --options`			Λίστα χωριζόμενη με κόμμα των επιλογών προσάρτησης.
`--block`	true, false	false	Εκτελεί το πρόγραμμα `mergerfs`, μπλοκάρει για σήματα `SIGINT` ή `SIGTERM` και αποπροσαρτά. Αν τεθεί σε false, εκτελεί το πρόγραμμα `mergerfs` μόνο.

Πίνακας 3.1: *Παράμετροι εντολής "gogomergerfs mergerfs"*

Εάν παρέχεται το όρισμα `--block`, το `gogomergerfs` εκτελεί την εντολή `mergerfs` και στη συνέχεια μπλοκάρει, περιμένοντας τα σήματα `SIGINT` ή `SIGTERM`. Όταν ληφθεί ένα τέτοιο σήμα, το `gogomergerfs` προσπαθεί να αποπροσαρτήσει το προκύπτον σύστημα αρχείων `mergerfs` από το σημείο προσάρτησης, προτού τερματίσει.

Με το Gogomergerfs, ο κύκλος ζωής του συστήματος αρχείων MergerFS (του τόμου του Union CSI) συγχρονίζεται με το κύκλο ζωής του Pod. Το σύστημα αρχείων παραμένει ενεργό, ενώ το Pod τρέχει και αποπροσαρτάται από τον οικοδεσπότη, όταν το container λαμβάνει σήμα `SIGTERM` για να τερματίσει. Το πρόγραμμα Gogomergerfs μπορεί να επεκταθεί με επιπρόσθετες λειτουργίες, όπως η παρακολούθηση του συστήματος αρχείων και η καταγραφή της κατάστασής του ενώ περιμένει για τα σήματα, διευκολύνοντας έτσι τον εντοπισμό τυχόν σφαλμάτων.

3.2.2 Χειριστής VolumeSplit

Το τμήμα του Χειριστή VolumeSplit αποτελείται κυρίως από τον ορισμό του τύπου του ειδικού πόρου VolumeSplit σε Go, το αρχείο manifest σε YAML του CRD του πόρου και έναν ειδικό χειριστή που χειρίζεται τα αντικείμενα VolumeSplits.

Kubebuilder SDK

Αυτό το τμήμα αναπτύχθηκε με τη βοήθεια του εργαλείου *Kubebuilder*. Το Kubebuilder είναι ένα πλαίσιο λογισμικού που βοηθά σημαντικά τους προγραμματιστές στη δημιουργία και δημοσίευση των Kubernetes APIs τους μέσω των CustomResourceDefini-

tions (CRDs), χρησιμοποιώντας τη γλώσσα Go και τη γραμμή εντολών.

Η ισχυρή του βιβλιοθήκη προσφέρει μια απλή διεπαφή, που βασίζεται σε και απλοποιεί τον παραδοσιακό τρόπο γραφής των Kubernetes APIs και των χειριστών—μια δύσκολη, κουραστική και επιρρεπής σε σφάλματα διαδικασία, ακόμα και για έμπειρους προγραμματιστές Kubernetes. Αυτό επιτρέπει στους χρήστες να επεκτείνουν γρήγορα και αποτελεσματικά το Kubernetes.

Η μέδοθος `Reconcile`

Σχετικά με τον χειριστή VolumeSplit, χρησιμοποιώντας τη βιβλιοθήκη `controller-runtime` του Kubebuilder, η λογική συμφιλίωσης συμπυκνώνεται σε μια μόνο συνάρτηση που ονομάζεται `Reconcile`. Εδώ, ο προγραμματιστής προσθέτει τη λογική του χειριστή. Η συνάρτηση `Reconcile` λαμβάνει το όνομα και τον χώρο ονομάτων (εάν το αντικείμενο ανήκει σε έναν) του αντικειμένου προς συμφιλίωση, ανακτά το αντικείμενο από την τοπική κρυφή μνήμη cache, εφαρμόζει τη λογική ελέγχου και επιστρέφει σφάλμα, εάν το αντικείμενο χρειάζεται να επαναδρομολογηθεί για συμφιλίωση.

Η Reconcile είναι μια μέθοδος ενός *reconciler*. Ένα αντικείμενο reconciler καταχωρείται σε ένα ”υψηλότερο” αντικείμενο, που ονομάζεται `Manager`, το οποίο παρέχεται από τη βιβλιοθήκη `controller-runtime`. Ο `Manager` χρησιμοποιείται για να καταχωρίσει και να διαχειριστεί πολλούς διαφορετικούς reconcilers, αρχικοποιώντας τις κρυφές μνήμες τους, τους πελάτες Kubernetes τους, τις ουρές και τις goroutines εργασίας και να πραγματοποιεί τις κλήσεις `Reconcile`. Ο `Manager` παρακολουθεί τον διακομιστή API για συμβάντα δημιουργίας, ενημέρωσης και διαγραφής σχετικά με τους εγγεγραμμένους πόρους API και καλεί την αντίστοιχη μέθοδο `Reconcile` για το τροποποιημένο αντικείμενο. Ένας reconciler μπορεί να ενημερώσει τον `Manager` για οποιουσδήποτε πόρους API του ανήκουν (μέσω των αναφορών `metadata.ownerReferences`), ώστε η `Reconcile` να κληθεί για το αντικείμενο ιδιοκτήτη για οποιεσδήποτε αλλαγές στα αντικείμενα που του ανήκουν.

Η δομή του reconciler για το API VolumeSplit ονομάζεται `VolumeSplitReconciler`. Η συνάρτηση `main` του τμήματος του Χειριστή VolumeSplit δημιουργεί ένα νέο αντικείμενο `Manager`, καταχωρεί ένα αντικείμενο `VolumeSplitReconciler` σε αυτό και ξεκινά το `Manager`.

Στη μέθοδο `Reconcile` του reconciler `VolumeSplitReconciler`, ανακτούμε το αντι-

κείμενο VolumeSplit που εξετάζεται και όλα τα ρέπλικά του (τα αντικείμενα PersistentVolumeClaim που ανήκουν στο VolumeSplit). Στη συνέχεια συγκρίνουμε τον αριθμό των ευρεθέντων ρέπλικα με τον καθορισμένο αριθμό στο πεδίο `spec.replicas` του VolumeSplit. Εάν τα ρέπλικα είναι λιγότερα από τον επιθυμητό αριθμό, δημιουργούμε περισσότερα χρησιμοποιώντας το πρότυπο PVC στο `spec.template` του VolumeSplit.

3.2.3 Union CSI

Το τμήμα Union CSI αποτελεί το κεντρικό μέρος του λογισμικού Union CSI και της διατριβής αυτής. Περιλαμβάνει τον οδηγό CSI, που υλοποιεί τις υπηρεσίες gRPC του CSI και ενσωματώνεται με το Kubernetes, και του backend αποθήκευσης, που καλείται από τον οδηγό CSI για την εκτέλεση λειτουργιών CRUD για τους τόμους Union CSI χρησιμοποιώντας το Kubernetes.

Οδηγός CSI

Το τμήμα του Union CSI το οποίο υλοποιεί τα υποστηριζόμενα RPCs, δημιουργεί και ”ακούει” σε ένα σημείο CSI και ανταποκρίνεται στα RPCs που καλούν οι βοηθητικοί containers και το Kubelet.

Οι οδηγίες σχεδίασης, οι υποστηριζόμενες δυνατότητες, τα υλοποιημένες RPCs και η ροή εκτέλεσης του οδηγού CSI αναλύονται εκτενώς και συζητιούνται στο Κεφάλαιο 2.

Οι παράμετροι της γραμμής εντολών της εφαρμογής Union CSI ρυθμίζουν κυρίως τον οδηγό CSI.

Η επιλογή `--mode` ελέγχει το σύνολο των υπηρεσιών RPC που προσφέρει ο οδηγός σε ένα τρέχον στιγμιότυπο. Η τιμή ”controller” ρυθμίζει τον οδηγό για να καταχωρίσει και να εξυπηρετήσει κλήσεις Controller και Identity, η ”node” για να εξυπηρετήσει κλήσεις Node και Identity, και η ”all” για να εξυπηρετήσει και τις τρεις κατηγορίες.

Backend αποθήκευσης

Αν και αναφέρεται ως *backend*, δεν παρέχει σημεία HTTP, ούτε αποκαλύπτει οποιουδήποτε είδους API. Είναι μια υλοποίηση της διεπαφής Go `StorageBackend` (όπως σχεδιάστηκε από εμάς), που χρησιμοποιείται από τον οδηγό CSI για να διαχειρίζεται τους πόρους VolumeSplit, PVC και Pod, αποκρύπτοντας οποιεσδήποτε σχετικές με το Kubernetes

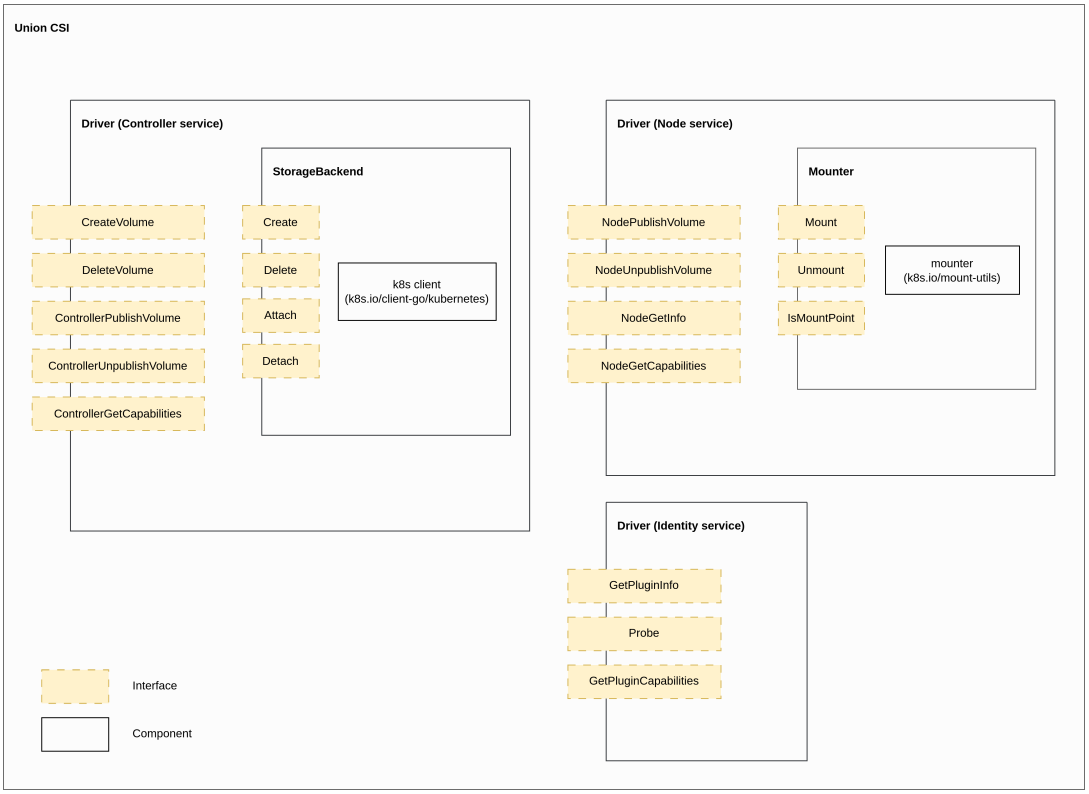
Παράμετρος	Τιμές	Προεπιλογή	Περιγραφή
`--endpoint`		”unix:///tmp/csi.sock”	Το σημείο gRPC του οδηγού CSI.
`--mode`	”controller”, ”node”, ”all”	”all”	Θέτει τη λειτουργία του οδηγού CSI. Η λειτουργία ”controller” τρέχει την υπηρεσία Controller, η ”node” την υπηρεσία Node και η ”all” και τις δύο μαζί.
`--kubeconfig`			Το απόλυτο μονοπάτι σε ένα αρχείο kubeconfig. Χρήσιμο σε λειτουργία εκτός συστοιχίας.
`-v, --version`			Τυπώνει την έκδοση και τερματίζει.

Πίνακας 3.2: *Παράμετροι εντολής ”union-csi”*

λεπτομέρειες από αυτόν. Ο οδηγός CSI χρησιμοποιεί το backend αποθήκευσης κατά τις κλήσεις RPC `CreateVolume`, `DeleteVolume`, `ControllerPublishVolume` και `ControllerUnpublishVolume`, προωθώντας το σχετικό αίτημα. Χρησιμοποιούμε τον όρο ”backend”, επειδή αυτή η διεπαφή λειτουργεί ως το ”αποθηκευτικό backend” ή την ”υπηρεσία αποθήκευσης”, που ο οδηγός CSI ενσωματώνει με το Kubernetes.

Η υλοποίηση ακολουθεί ακριβώς τον σχεδιασμό και τους αλγορίθμους που περιγράφονται στην Ενότητα 2.6.

Για παράδειγμα, στο RPC `CreateVolume`, ο οδηγός CSI καλεί τη μέθοδο `Create` του backend, χρησιμοποιώντας το `volume_id` μαζί με άλλες τιμές που λαμβάνονται από το `CreateVolumeRequest`. Στη μέθοδο `Create`, το backend ελέγχει πρώτα αν ένας τόμος με το δοσμένο αναγνωριστικό υπάρχει ήδη: κάνει μια αίτηση `GET` στον εξυπηρετητή API του Kubernetes για ένα αντικείμενο VolumeSplit στο χώρο ονομάτων `union` με το δοσμένο αναγνωριστικό ως όνομα. Εάν το VolumeSplit υπάρχει, η πληροφορία του πεδίου `spec` συγκρίνεται με τη διαβιβαζόμενη πληροφορία, για να εξασφαλιστεί η ταυτοδυναμία, επιστρέφοντας σφάλμα αν δεν είναι συμβατές. Αν το αντικείμενο VolumeSplit δεν υπάρχει, τότε το backend πραγματοποιεί μια αίτηση `CREATE` στο Kubernetes, για να δημιουργήσει ένα νέο VolumeSplit. Σε κάθε επιτυχημένη περίπτωση, το backend μπλοκάρει για μια σύντομη περίοδο, περιμένοντας το αντικείμενο VolumeSplit να μεταβεί σε κατάσταση ετοιμότητας. Εάν η διαδικασία αναμονής επιτύχει, το backend επιστρέ-



Σχήμα 3.1: *Η δομή του Union CSI*

φει μια εσωτερική αναπαράσταση του VolumeSplit στον οδηγό και αυτός επιστρέφει επιτυχώς με τη σειρά του.

4 Αξιολόγηση

Σε αυτό το κεφάλαιο, δοκιμάζουμε το Union CSI και αποκτούμε μία πρώτη κατανόηση για την απόδοσή του.

4.1 Επισκόπηση

Θα αξιολογήσουμε το πρόσθετο Union CSI, μαζί με καθιερωμένα πρόσθετα τόμων για το Kubernetes, που αξιοποιούν την τοπική αποθήκευση σε κάθε κόμβο για την παροχή μόνιμων τόμων σε Pods.

Για να θέσουμε ένα σημείο αναφοράς, χρησιμοποιούμε το *Local Path Provisioner*, διεξάγοντας δοκιμές σε απλούς καταλόγους του οικοδεσπότη. Επιπλέον, χρησιμοποιούμε το *Longhorn*, ένα σύστημα αποθήκευσης μπλοκ επιχειρησιακού επιπέδου, ως το κατώτερο σύστημα αποθήκευσής μας.

Στις δοκιμές μας, εξετάζουμε αυτά τα δύο πρόσθετα ατομικά, καθώς επίσης και σε συνδυασμό με το Union CSI. Αξιολογούμε μετρήσεις, όπως ρυθμό μετάδοσης δεδομένων και καθυστέρηση, σε μία προσπάθεια να κατανοήσουμε την επίδραση του επιπέδου FUSE, που εισάγεται από την χρήση του MergerFS πάνω στους υποκείμενους τόμους.

4.2 Περιγραφή των Πρόσθετων Τόμων

Σε αυτήν την ενότητα, παρέχουμε μια επισκόπηση του πρόσθετου Local Path Provisioner και του συστήματος αποθήκευσης Longhorn, εξηγώντας τον σχεδιασμό τους, την αρ-

χιτεκτονική τους και τη λειτουργικότητά τους.

4.2.1 Local Path Provsioner

Το Local Path Provisioner, ανεπτυγμένο από την Rancher Labs, είναι ένα πρόσθετο τόμων για το Kubernetes, σχεδιασμένο να χρησιμοποιεί την τοπική αποθήκευση του κάθε κόμβου. Παρέχει δυναμικά είτε `hostPath`, είτε `local` μόνιμους τόμους.

Το Local Path Provisioner απαιτεί πληροφορία σχετική με τον κόμβο που θα δημιουργηθεί ο κατάλογος `hostPath`. Ως αποτέλεσμα, το Local Path Provisioner λειτουργεί αποκλειστικά με ένα StorageClass καθυστερημένης (`WaitForFirstConsumer`) λειτουργίας δέσμευσης τόμου. Επιπλέον, δεδομένου ότι οι τόμοι τύπου `hostPath` είναι τοπικά περιορισμένοι, το σχετικό PV, που δημιουργείται από το Local Path Provisioner, διαθέτει περιορισμό κόμβου (node affinity). Αυτό εξασφαλίζει, ότι τα Pods που χρησιμοποιούν το PV θα χρονοδρομολογηθούν μόνο στον κόμβο όπου αρχικά δημιουργήθηκε ο κατάλογος `hostPath`.

Λόγω των πολλών κινδύνων ασφάλειας και της μη ευελιξίας των τόμων τύπου `hostPath`, το Local Path Provisioner προορίζεται κυρίως για τοπικές συστοιχίες και περιβάλλοντα δοκιμών.

4.2.2 Longhorn

Το Longhorn είναι ένα δωρεάν, ανοικτού κώδικα, cloud-native, κατανεμημένο σύστημα αποθήκευσης μπλοκ, σχεδιασμένο για το Kubernetes. Χρησιμοποιεί την τοπική αποθήκευση των κόμβων του Kubernetes, εξαλείφοντας την ανάγκη για εξωτερικές εγκαταστάσεις αποθήκευσης και παροχείς νέφους. Το Longhorn χρησιμοποιεί containers, microservices και CRDs για την υλοποίηση κατανεμημένης αποθήκευσης μπλοκ.

Κάθε τόμος στο Longhorn αντιμετωπίζεται ως ένα microservice, με έναν αφιερωμένο χειριστή αποθήκευσης, που δημιουργείται για κάθε τόμο που ανήκει σε συσκευή μπλοκ. Αυτός ο χειριστής αποθήκευσης, γνωστός ως *Longhorn Engine*, τρέχει ως μία διεργασία Linux στον ίδιο κόμβο, όπου χρησιμοποιείται ο τόμος Longhorn από ένα Pod. Το Longhorn Engine αναπαράγει τον τόμο ασύγχρονα σε πολλαπλά αντίγραφα, που διανέμονται και αποθηκεύονται σε διαφορετικούς κόμβους. Αυτά τα αντίγραφα,

που αποτελούν επίσης microservices, αποτελούνται από αραιά αρχεία (sparse files) του Linux.

Το Longhorn δεν είναι ένα σύστημα με γνώση περί τοπολογίας, που σημαίνει ότι δεν απαιτεί γνώση του κόμβου χρονοδρομολόγησης ενός Pod για να ξεκινήσει την παροχή ενός τόμου για αυτόν. Ως αποτέλεσμα, το παρεχόμενο Longhorn StorageClass ορίζεται από προεπιλογή στη λειτουργία δέσμευσης τόμου `Immediate` και οι δημιουργηθέντες τόμοι και PVs δεν διαθέτουν περιορισμούς κόμβου.

Επιπλέον, για τη χρήση των τόμων από Pods σε οποιουσδήποτε κόμβους της συστοιχίας, το Longhorn χρησιμοποιεί το iSCSI (Open-iSCSI). Εξ' ορισμού, το Longhorn χρησιμοποιεί το δίκτυο Container Network Interface (CNI) της συστοιχίας Kubernetes.

Το Longhorn αναπτύχθηκε αρχικά από την Rancher Labs, ενώ τώρα αναγνωρίζεται ως ένα έργο που ανήκει στο Cloud Native Computing Foundation (CNCF).

4.3 Μετρήσεις Απόδοσης

Σε αυτήν την ενότητα, παρουσιάζουμε σχετικά αποτελέσματα απόδοσης, όσον αφορά τον μετρούμενο ρυθμό μετάδοσης δεδομένων και την καθυστέρηση, για τα τρία συστήματα αποθήκευσης Local Path Provisioner, Longhorn και Union CSI, σε διάφορα σενάρια δοκιμών. Μέσω αυτών των δοκιμών, στοχεύουμε στο να κατανοήσουμε τον αντίκτυπο των επιπέδων MergerFS και δικτύου στα υποκείμενα συστήματα αρχείων.

4.3.1 Περιβάλλον

Τα πρόσθετα Local Path Provisioner, Longhorn και Union CSI εγκαθίστανται και δοκιμάζονται σε μία συστοιχία Kubernetes των δύο κόμβων, που διαχειρίζεται η υπηρεσία AKS (Azure Kubernetes Service) της Azure. Οι κόμβοι είναι εικονικές μηχανές γενικής χρήσης της βαθμίδας `Standard_DS2_v2` [1], εξοπλισμένες η κάθε μία με 2 πυρήνες vCPU, 7GiB RAM και ένα τοπικό SSD χωρητικότητας 128GiB. Κάθε κόμβος εξυπηρετεί διπλούς ρόλους, ως κύριος (control-plane) και ως κόμβος εργασίας στο Kubernetes.

[1] https://azureprice.net/vm/Standard_DS2_v2

4.3.2 Σενάρια Δοκιμών

Τα σενάρια δοκιμών περιλαμβάνουν τη μέτρηση της απόδοσης Ε/Ε των τόμων, που παρέχονται από τα πρόσθετα Local Path Provisioner, Longhorn και Union CSI. Κάθε πρόσθετο ελέγχεται μεμονωμένα ή σε συνδυασμό με το Union CSI, όπου ο δοκιμαζόμενος τόμος αποτελείται από το αποτέλεσμα του `mergerfs` σε δύο ίσους σε μέγεθος τόμους, που παρέχονται από το κατώτερο πρόσθετο. Οι παράμετροι του προγράμματος `mergerfs` αφέθηκαν στις προεπιλεγμένες τιμές τους. Για το Longhorn, οι δοκιμές πραγματοποιούνται τόσο με, όσο και χωρίς την παρεμβολή του δικτύου με την ανάθεση του Pod αξιολόγησης στον αντίθετο ή στον ίδιο κόμβο με αυτόν στον οποίο έχει δημιουργηθεί ο τόμος Longhorn.

Σε κάθε σενάριο δοκιμής, εκκινούμε ένα Pod που τρέχει ένα container με το εργαλείο αξιολόγησης συστημάτων αρχείων, που εκτελεί το καθορισμένο φορτίο Ε/Ε στον δοκιμαζόμενο τόμο. Οι "ετικέτες" και οι περιγραφές κάθε σεναρίου δοκιμής παρέχονται στις ακόλουθες παραγράφους.

Local: Για να αποκτήσουμε ένα πρότυπο για σύγκριση, είναι σημαντικό να μετρήσουμε την απόδοση ανάγνωσης και εγγραφής σε έναν απλό κατάλογο στο σύστημα αρχείων του κόμβου. Το Local Path Provisioner δημιουργεί έναν κατάλογο στον κόμβο του οικοδεσπότη, που είναι προσαρτημένος στο container, που περιέχει το εργαλείο μας για τη μέτρηση της απόδοσης Ε/Ε σε αυτό το σενάριο. Αναμένεται ότι αυτό το σενάριο θα αποφέρει τις υψηλότερες μετρήσεις σε Ε/Ε.

Longhorn: Ρυθμίζουμε το Longhorn, ώστε να δημιουργεί μόνο έναν κύριο αντίγραφο κατά τη δημιουργία ενός PVC. Το τόμος εκχωρείται ως αραιό αρχείο σε έναν από τους δύο κόμβους. Σε αυτό το σενάριο, μετράμε την απόδοση ενός τόμου Longhorn, που έχει δημιουργηθεί στον ίδιο κόμβο με αυτόν που χρονοδρομολογείται το Pod. Έτσι θα αναδειχθεί το σύστημα μπλοκ του Longhorn στο έπακρο.

Longhorn + iSCSI: Ίδιο σενάριο με πριν, αλλά τώρα περιορίζουμε το Pod μας που πραγματοποιεί την αξιολόγηση, ώστε να τρέξει στον αντίθετο κόμβο (προσθέτοντας την κατάλληλη τιμή `nodeSelector`), εκθέτοντας τη συσκευή μπλοκ του Longhorn σε αυτό μέσω iSCSI. Με αυτόν τον τρόπο, μπορούμε να ποσοτικοποιήσουμε την προστιθέμενη καθυστέρηση του δικτύου του SCSI μέσω Ethernet στη συστοιχία μας με δύο κόμβους.

Union + Local: Η χρήση του Local Path Provisioner ως υποκείμενου προμηθευτή του

Union CSI δεν αποφέρει κάποιο όφελος. Οι δύο κατώτεροι τόμοι `hostPath` θα δημιουργηθούν αναγκαστικά στον ίδιο κόμβο, οπότε δεν υπάρχει τρόπος να εκμεταλλευτούμε τον χώρο δίσκου από διαφορετικούς κόμβους με το Union CSI. Ωστόσο, με τη χρήση απλών καταλόγων στον οικοδεσπότη για τους κλάδους του MergerFS, μπορούμε να μετρήσουμε την επίδραση του στρώματος του συστήματος αρχείων FUSE απευθείας στο σύστημα αρχείων του κεντρικού υπολογιστή.

Union + Longhorn: Σε αυτό το σενάριο, χρησιμοποιούμε το Longhorn ως τον κατώτερο προμηθευτή CSI για το Union CSI. Ωστόσο, θέλουμε οι τόμοι Longhorn να δημιουργηθούν στον ίδιο κόμβο. Για αυτόν τον σκοπό, δημιουργούμε έναν αρκετά μικρό τόμο Union CSI που να χωράει στον δίσκο του κόμβου, όπως για παράδειγμα έναν τόμο 30GiB διαιρεμένο σε δύο τόμους Longhorn των 15GiB ο καθένας. Στη συνέχεια, αναθέτουμε το Pod αξιολόγησης στον ίδιο κόμβο με τους κατώτερους τόμους για να μετρήσουμε την απόδοση της προσάρτησης union δύο τοπικών τόμων Longhorn.

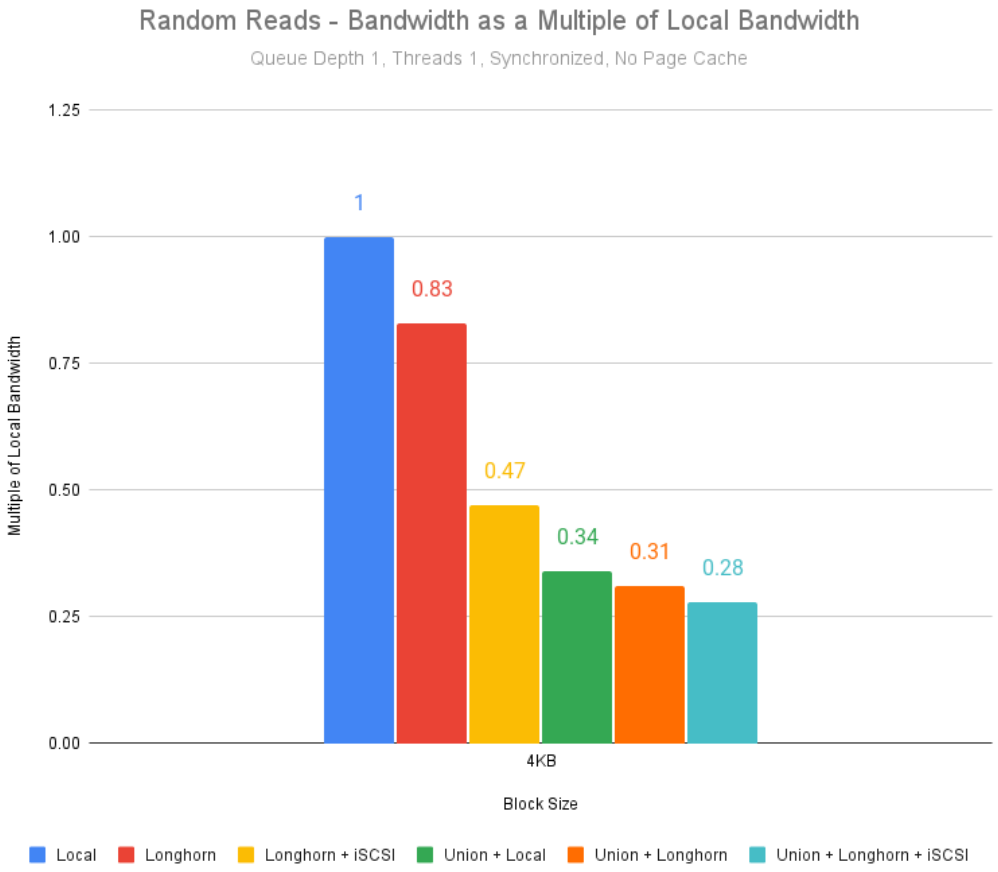
Union + Longhorn + iSCSI: Ίδιο σενάριο με το προηγούμενο, αλλά τώρα το Pod ανατίθεται στον αντίθετο κόμβο από αυτόν στον οποίο έχουν προγραμματιστεί οι δύο τόμοι Longhorn. Αυτό σημαίνει, ότι οποιαδήποτε αιτούμενη E/E πρέπει να περάσει μέσω του επιπέδου FUSE, της υλοποίησης iSCSI και του δικτυακού, και του επιπέδου συσκευών μπλοκ του Longhorn. Αυτό αντιπροσωπεύει ένα πιο ρεαλιστικό σενάριο για το Union CSI, όπου τουλάχιστον ένας από τους κλάδους θα είναι συνδεδεμένος απομακρυσμένα.

4.3.3 Αποτελέσματα

Τα αποτελέσματα του ρυθμού μετάδοσης δεδομένων και της καθυστέρησης που μετρήθηκαν είναι κανονικοποιημένα ως προς το σενάριο δοκιμής *Local*—όλες οι τιμές εμφανίζονται ως πολλαπλάσια της τιμής του *Local*

Η Εικόνα 4.1 παρουσιάζει την απόδοση του ρυθμού μετάδοσης δεδομένων τυχαίων αναγνώσεων για κάθε περίπτωση δοκιμής, χρησιμοποιώντας μέγεθος μπλοκ 4KiB. Όπως αναμενόταν, το *Local* εμφανίζει τη μεγαλύτερη απόδοση μεταξύ όλων των διαγωνιζομένων. Ακολουθεί κοντά το *Longhorn*, με μείωση 17% στην απόδοση τυχαίων αναγνώσεων, η οποία πέφτει περαιτέρω στο 47% για το *Longhorn + iSCSI* λόγω της πρόσθετης καθυστέρησης του SCSI και του δικτύου. Στις δοκιμές *Union*, και οι τρεις περιπτώσεις παρείχαν πανομοιότυπα αποτελέσματα, που διακυμαίνονται γύρω στο 30% της από-

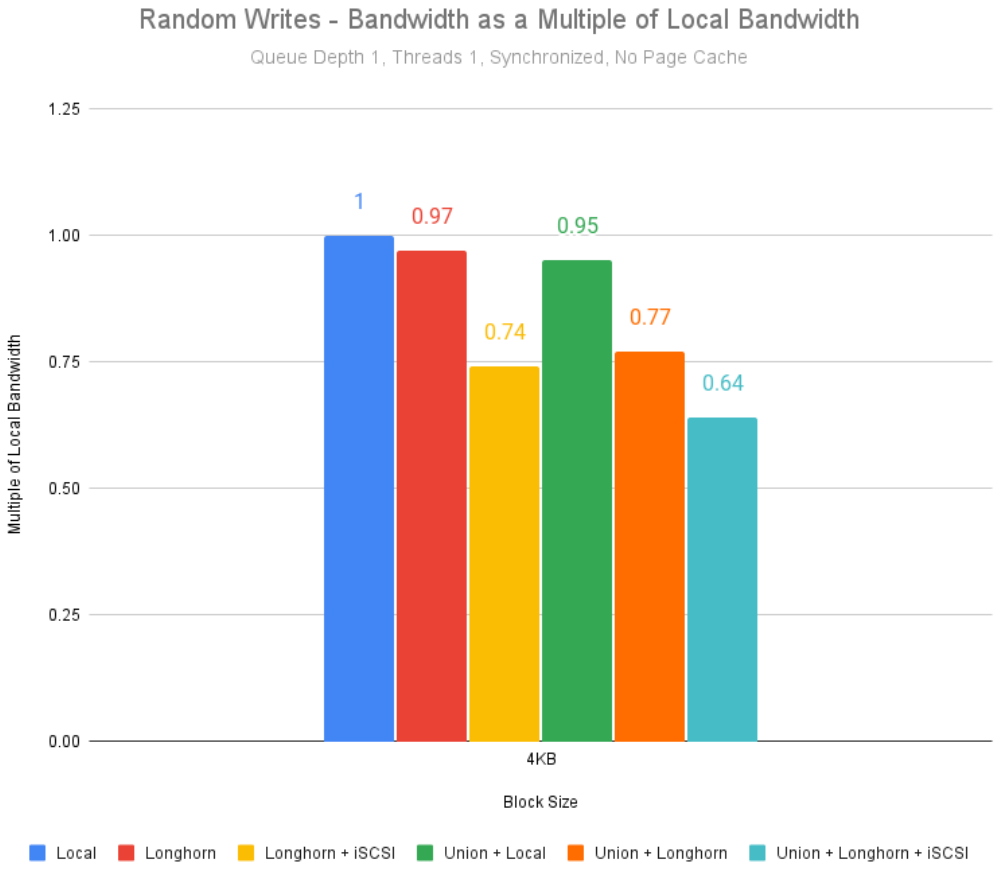
δοσης του *Local*. Τα αποτελέσματα τυχαίων αναγνώσεων του Union CSI, παρόλο που είναι τα χαμηλότερα μεταξύ όλων, δεν υπέστησαν περαιτέρω υποβάθμιση, είτε χρησιμοποιώντας απλούς καταλόγους, είτε την αποθήκευση επιπέδου μπλοκ του Longhorn με απομακρυσμένη πρόσβαση.



Σχήμα 4.1: *Απόδοση ρυθμού μετάδοσης δεδομένων τυχαίων αναγνώσεων για μέγεθος μπλοκ 4KiB*

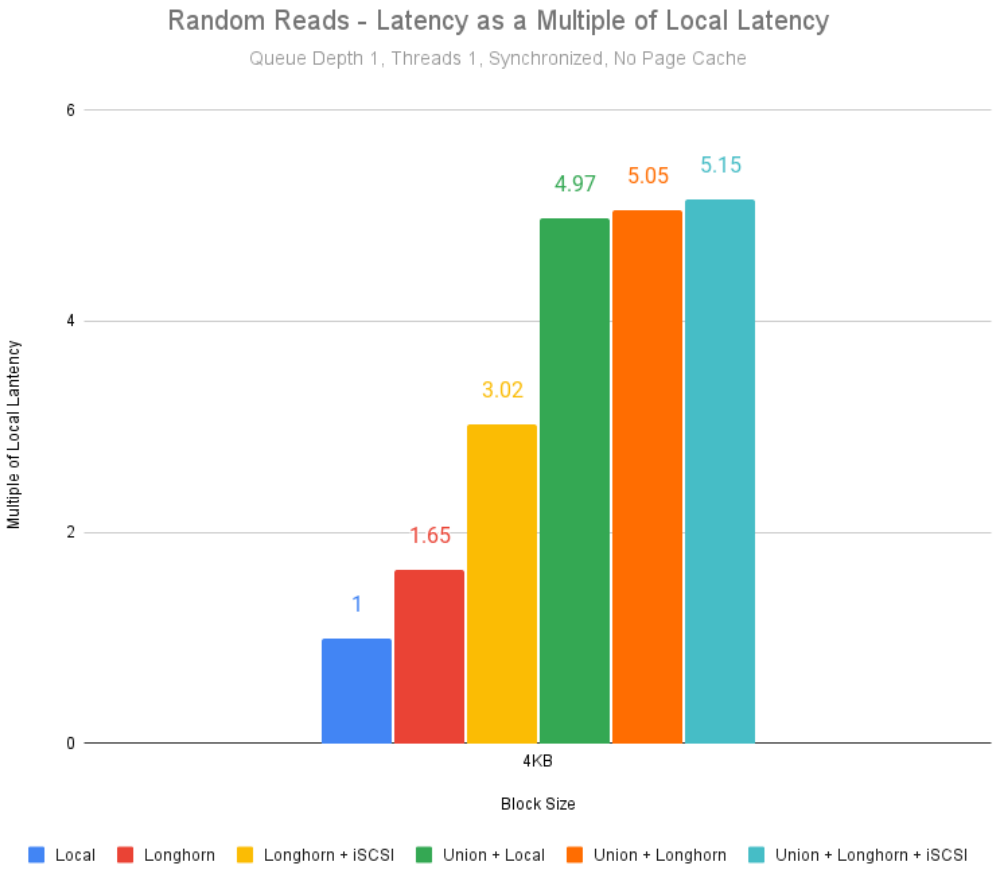
Η Εικόνα 4.2 παρουσιάζει την απόδοση του ρυθμού μετάδοσης δεδομένων τυχαίων εγγραφών για κάθε περίπτωση δοκιμής, χρησιμοποιώντας μέγεθος μπλοκ 4KiB. *Longhorn* και *Union + Local* σχεδόν πετυχαίνουν το όριο του *Local*, ενώ τα δύο σενάρια με τους απομακρυσμένους τόμους, *Longhorn + iSCSI* και *Union + iSCSI*, έχουν τις χαμηλότερες βαθμολογίες στο 74% και 64% αντίστοιχα. Στις περιπτώσεις *Union*, η αντικατάσταση του Local Path Provisioner με το Longhorn, ως τον κατώτερο προμηθευτή (*Union + Longhorn*), οδηγεί σε μείωση της απόδοσης κατά περίπου 19%, με περαιτέρω μείωση κατά 17% σε σύγκριση με το *Union + Longhorn*, όταν πραγματοποιείται μεταφορά

δεδομένων μέσω του δικτύου (*Union* + *Longhorn* + *iSCSI*).



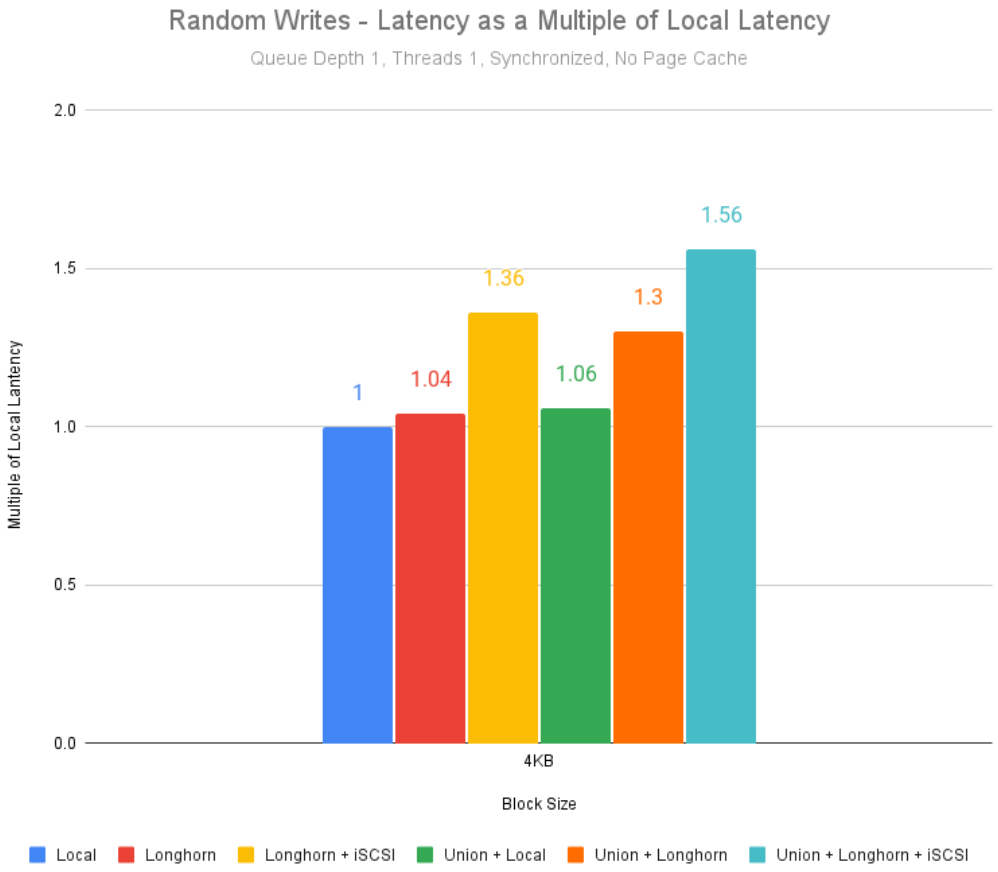
Σχήμα 4.2: *Απόδοση ρυθμού μετάδοσης δεδομένων τυχαίων εγγραφών για μέγεθος μπλοκ 4KiB*

Η Εικόνα 4.3 παρουσιάζει την απόδοση της καθυστέρησης τυχαίων αναγνώσεων για κάθε περίπτωση δοκιμής, χρησιμοποιώντας μέγεθος μπλοκ 4KiB. Παρατηρούμε ότι, στο περιβάλλον μας και την παρούσα διαμόρφωση, η τυχαία ανάγνωση από έναν τοπικό τόμο Longhorn είναι 1,65 φορές πιο αργή από το *Local*, ενώ από έναν απομακρυσμένο τόμο Longhorn είναι μέχρι και 3 φορές πιο αργή. Παρόμοια με τα αποτελέσματα των τυχαίων αναγνώσεων ρυθμού μετάδοσης δεδομένων, οι δοκιμές του *Union* εμφανίζουν όλες παρόμοιο επίπεδο απόδοσης, περίπου 5 φορές πιο αργές από το *Local*.



Σχήμα 4.3: *Απόδοση καθυστέρησης τυχαίων αναγνώσεων για μέγεθος μπλοκ 4KiB*

Η Εικόνα 4.4 παρουσιάζει την απόδοση της καθυστέρησης τυχαίων εγγραφών για κάθε περίπτωση δοκιμής, χρησιμοποιώντας μέγεθος μπλοκ 4KiB. Όπως και με τα αποτελέσματα των τυχαίων αναγνώσεων ρυθμού μετάδοσης δεδομένων, το *Longhorn* και το *Union + Local* επιτυγχάνουν πρακτικά την ίδια απόδοση με το *Local*. Η χρήση τοπικών τόμων Longhorn με το Union CSI είναι 1,3 φορές πιο αργή από το *Local*, ενώ οι περιπτώσεις που περιλαμβάνουν το δίκτυο είναι οι πιο αργές, με το *Union + Longhorn + iSCSI* να είναι 1,56 φορές πιο αργό από το *Local*.



Σχήμα 4.4: *Απόδοση καθυστέρησης τυχαίων εγγραφών για μέγεθος μπλοκ 4KiB*

4.3.4 Σύνοψη

Η σχετικά χαμηλότερη απόδοση που παρατηρήθηκε για τους Union CSI τόμους ήταν αναμενόμενη, λόγω του επιπέδου MergerFS/FUSE πάνω από τους υποκείμενους τόμους, ιδιαίτερα στην περίπτωση όπου η μεταφορά δεδομένων πραγματοποιήθηκε μέσω iSCSI για τους τόμους του Longhorn.

Το MergerFS είναι ένα πρόγραμμα χώρου χρήστη FUSE που λειτουργεί μεσολαβητής για τα υποκείμενα συστήματα αρχείων. Οι αιτήσεις Ε/Ε στο σύστημα αρχείων FUSE συνεπάγονται πρόσθετη καθυστέρηση λόγω των μεταβάσεων μεταξύ χώρου χρήστη και χώρου πυρήνα. Ένας τρόπος για να μειωθεί ο αριθμός των μεταβάσεων είναι η αύξηση του μεγέθους του μπλοκ Ε/Ε. Στις δοκιμές μας, ορίσαμε ένα αρκετά μικρό μέγεθος μπλοκ 4KiB για να επικεντρωθούμε σε τυχαία Ε/Ε. Η χρήση μεγαλύτερου μεγέθους μπλοκ, μαζί με την ενεργοποίηση του page caching του λειτουργικού συστήματος, θα

μπορούσε να βελτιώσει σημαντικά την απόδοση του Union CSI, ακόμα και σε σχετική αύξηση μεγαλύτερη από τα άλλα πρόσθετα.

Ακόμα, ο συγγραφέας του MergerFS επισημαίνει διάφορους λόγους για τη μειωμένη απόδοση σε προσαρτήσεις MergerFS και παρέχει εισηγήσεις για τεχνικές μέτρησης απόδοσης και ρυθμίσεις παραμέτρων [2]. Το MergerFS προσφέρει πολλές ρυθμίσιμες επιλογές, πολλές από τις οποίες επηρεάζουν την απόδοση. Μια πιο βαθιά κατανόηση αυτών των επιλογών και η λεπτομερής ρύθμισή τους θα μπορούσε να οδηγήσει σε καλύτερα αποτελέσματα από εκείνα που παρουσιάζονται στις δοκιμές μας.

Συνοψίζοντας, ακόμη και η πιο εξελιγμένη εφαρμογή χώρου χρήστη έχει περιορισμούς στη διατήρηση αυξημένης απόδοσης και στην ελαχιστοποίηση των μεταβάσεων χρήστη-πυρήνα. Τα αποτελέσματα των δοκιμών μας υπογραμμίζουν την ανάγκη για μια λύση ενώσεως αρχείων, βασισμένη στον πυρήνα για φορτία εργασιών αποθήκευσης επιχειρηματικού επιπέδου.

[2] https://github.com/trapexit/mergerfs#performance

5 Επίλογος

Πλησιάζουμε τις τελικές σελίδες αυτής της διατριβής. Σε αυτό το κεφάλαιο, συνοψίζουμε τα κίνητρα και τα κύρια στοιχεία που παρουσιάζονται στη διατριβή μας. Στη συνέχεια, περιγράφουμε τη λίστα μελλοντικών προσθηκών μας, κατανοώντας την τρέχουσα θέση μας και θέτοντας νέους στόχους.

5.1 Συμπερασματικά Σχόλια

Ο κύριος στόχος αυτής της διατριβής και του συναφούς έργου της ήταν η εξερεύνηση και υλοποίηση μιας καινοτόμου λύσης αποθήκευσης. Αυτή η λύση στοχεύει στο να υποστηρίξει συστήματα αρχείων πολλαπλών terabytes συνδυάζοντας μικρότερα αποθηκευτικά τμήματα, ξεπερνώντας τα όρια χωρητικότητας και τοπολογίας των υπαρχόντων συστοιχιών τοπικής αποθήκευσης.

Αρχικά, προτείναμε το Union CSI ως ένα ”μέτα-πρόσθετο”, λειτουργώντας σε συνεργασία με άλλα πρόσθετα αποθήκευσης για να αθροίζει τους τόμους τους. Δόθηκε μια εκτενής ανάλυση για τις βασικές σχεδιαστικές έννοιες, την αρχιτεκτονική και τις αποφάσεις που διέπουν το Union CSI, καθιστώντας το μια κατάλληλη λύση συμβατή με το Kubernetes και το CSI που παρέχει αποθήκευση, ως προσαρτήσεις union των συστημάτων αρχείων του παρόχου. Συνεχίσαμε με την περιγραφή της υλοποίησης των διαφόρων συνιστωσών του Union CSI. Έπειτα, παρουσιάσαμε μια σύγκριση των σχετικών επιδόσεων του Union CSI μαζί με το Longhorn. Ενώ τα αποτελέσματα των δοκιμών δεν ήταν υπέρ του Union CSI—και πιθανότατα οι πραγματικές εργασίες E/E δεν θα

επέφεραν τέτοια περιορισμένα αποτελέσματα—μπορούμε τώρα να υποστηρίξουμε με αυτοπεποίθηση την περίληψη του MergerFS στον πυρήνα του Linux.

Η ανάπτυξη του έργου αυτού ήταν μια ιδιαίτερα ενδιαφέρουσα και διαφωτιστική εμπειρία για τους εμπλεκόμενους. Μέσω αυτής της εργασίας, επιδιώκουμε να μεταφέρουμε κάποια από τη γνώση και τον ενθουσιασμό που αποκτήθηκαν.

5.2 Μελλοντικό Έργο

Ολοκληρώνοντας αυτήν την αρχική, κυρίως πειραματική και εκπαιδευτικού χαρακτήρα έκδοση του λογισμικού Union CSI, μαζί με την αξιολόγηση της γνώσης και της εμπειρίας που αποκτήθηκε, διαθέτουμε μια σαφή εικόνα για πιθανές βελτιώσεις. Εκτός από την αντιμετώπιση συγκεκριμένων περιπτώσεων (π.χ. ενίσχυση του χρόνου διαθεσιμότητας (uptime) των προσαρτήσεων MergerFS με χρήση Deployment ή StatefulSet αναπτύξεων αντί για απλά Pods, εμπλούτιση το API VolumeSplit ώστε να προσφέρει περισσότερες λεπτομέρειες σχετικά με τις αντίστοιχες καταστάσεις των ρέπλικα PVCs) και δαπανώντας περισσότερο χρόνο σε δοκιμές, αρκετές νέες κατευθύνσεις σχεδιασμού και υλοποίησης εμφανίζονται, συμπεριλαμβανομένων:

- Να κάνουμε το Union CSI να υπολογίζει τον διαθέσιμο χώρο δίσκου σε κάθε κόμβο, επιτρέποντας έξυπνες αποφάσεις για τον διαχωρισμό της αρχικής χωρητικότητας.
- Να εξελίξουμε το Union CSI σε ένα αυτόνομο σύστημα αποθήκευσης. Προς το παρόν, το Union CSI εξαρτάται από το υποκείμενο αποθηκευτικό σύστημα για το μονοπάτι δεδομένων και την ικανότητα αποθήκευσης μέσω του δικτύου, χωρίς άμεσα μέσα για να παρατηρήσει τα σχετικά αποτελέσματα εκτός από το Kubernetes API. Αυτή η επιλογή σχεδιασμού ήταν κυρίως επηρεασμένη από την ανάγκη για την έγκαιρη ολοκλήρωση αυτής της διπλωματικής εργασίας και, ομολογουμένως, είναι παράξενη, μη ανθεκτική και ευάλωτη σε πολλαπλά σημεία αποτυχίας. Οι ιδέες και τα πλεονεκτήματα του Union CSI, καθώς και των συστημάτων αρχείων union γενικότερα, θα πρέπει να ενσωματωθούν σε μια μεγαλύτερη, πλούσια σε χαρακτηριστικά, αυτόνομη λύση διαχείρισης αποθήκευσης.
- Να υποβάλλουμε το MergerFS σε δοκιμές αντοχής, συνδυάζοντας και οργανώνοντας εκατοντάδες τόμους. Τέλος, να δημιουργήσουμε μια proof-of-concept

υλοποίηση του MergerFS στον πυρήνα του Linux, να συγκρίνουμε την απόδοσή της με την υπάρχουσα υλοποίηση στο χώρο του χρήστη, και να χρησιμοποιήσουμε τα αποτελέσματα για να υποστηρίξουμε την ενσωμάτωσή της στον κύριο πυρήνα του Linux.

1 Introduction

In this opening chapter, we describe the scope of our work and present an outline for the structure of the remainder of the thesis.

1.1 Motivation

Storage is the foundation of modern data center architectures and stateful applications. High-capacity, high-performance, low-latency, and fault-tolerant storage is a crucial element for infrastructures and companies that handle mission-critical workloads with heavy data demands.

Cloud and storage providers have evolved to cater to a wide variety of compute and storage needs, and at the same time establishing a lucrative market. Meanwhile, the rapid evolution of container orchestration systems (COs), epitomized by Kubernetes, has revolutionized the deployment and management of containerized workloads. As organizations increasingly embrace containerization for their applications, where every stateful container comes with its own data store(s), the demand for efficient and scalable storage solutions within Kubernetes clusters has become paramount.

However, whether companies opt for on-premise or cloud-hosted infrastructures, incorporating Kubernetes or not, the age-old deployment dilemma remains: local or shared storage? Local or directly-attached storage (DAS) offers increased IOPS and throughput but comes with limited flexibility and scalability options. On the other hand, shared or remote storage (e.g., EBS on Amazon Web Services, Azure Disk Storage, Persistent Disks on Google Cloud Platform) is really flexible, facilitating snapshots, backups, and

node migrations, but limits performance and drives up costs.

Even though local storage has the clear performance and price lead, and most applications would be greatly accelerated if run over performant, local NVMe SSDs, many companies seem hesitant or are unable to make the leap to local storage because they are ultimately dependent on shared storage's flexibility. Having one's data secure, highly available and portable is crucial. But so are microsecond IO latency times and reduced price bills.

Additionally, in today's landscape characterized by the proliferation of Big Data, MLOps (Machine Learning Operations), and AI's LLMs (Large Language Models) consuming massive datasets—often in the scale of petabytes—and training times that stretch over months, the demand for super-fast, local SSDs at scale and within budget is dire as ever.

Is it possible for stateful applications to run over local NVMe SSDs while retaining all the advantages of shared storage? Are local and shared storage inherently mutually exclusive? We believe that the above dilemma no longer holds today. Modern applications should not have to trade performance for flexibility and vice versa. A modern enterprise storage and data management system should leverage local storage through a lightweight critical IO path while providing advanced storage capabilities and cloud-native data services, just like shared storage. Such a system would enable data scientists and developers to rapidly prototype, test, and deploy their applications and machine learning models, and companies to boost their performance, reduce storage costs, and unlock new business potential.

After all, enterprise production workloads demand enterprise solutions.

Motivated by this strong belief in a universal storage solution capable of combining the best of both worlds, in this dissertation, we explore a piece of this puzzle: scalability. We examine a simple way of aggregating large, multi-terabyte, persistent storage pools over high-performance, node-local disks in Kubernetes clusters.

1.2 Problem Statement

Local and shared storage are the two predominant, complementary storage accessing options. The former provides high performance but reduced flexibility, while the latter

is highly flexible but can compromise performance.

Cloud providers offer both of these options, along with in-between hybrid solutions. The current norm is that local storage serves as the cost-effective, entry-level tier, and shared storage expansibility is unlocked through premium tiers and corresponding increased costs.

Now, let's consider a scenario where a team of developers or application owners is running a Kubernetes cluster in the cloud. They have chosen to utilize the local disk block storage tied to each node in the cluster to benefit from its increased IOPS and low latency, using an appropriate persistent storage volume plugin.

However, what happens when the application's data is growing large and is running out of capacity on one or more nodes? What options are available? One option is to switch to the cloud supplier's shared storage solution to scale out by attaching a storage device from an external array. While the provided premium tiers may offer adequate performance, they come with increased pricing. Another option is to upgrade a subset of node machines to higher-capacity local devices, attempting to mimic a scale-up architecture. However, migrating the entire application and its data can be challenging, frustrating, and sometimes not even possible due to local storage restrictions.

Meanwhile, the remaining node storage within the cluster might be sitting there, perfectly free and available.

Is there an option to combine the cluster node storage into a single hierarchy, effectively bringing together all storage spaces available to each individual node into a single storage pool and presenting it to the user as a unified, cohesive view?

In this dissertation, we present a mechanism for aggregating storage spaces from different devices and nodes in the cluster to provide volumes that achieve super-node capacities.

1.3 Existing Solutions

There are several sophisticated and mature storage systems and platforms capable of consolidating the entire cluster block storage into a unified storage pool, accompanied by advanced features such as data replication, distributed storage, snapshotting, and

more.

One widely used example is Ceph, a free and open-source storage platform that provides distributed object, block, and file storage. Ceph organizes storage into pools that can span across different nodes in the cluster. When creating a volume or object in Ceph, the data is striped, replicated, and distributed across multiple *Object Storage Daemons* (OSDs) on various nodes, ensuring high availability and fault tolerance. Ceph utilizes commodity hardware and Ethernet IP, requiring no specialized equipment, to deliver scalable, resilient, self-healing, and self-managed distributed storage.

Ceph is a behemoth of a software-defined storage system with substantial resource requirements. It is highly demanding in terms of storage and compute resources, necessitating a significant number of disks, CPU, and RAM in order to support data replication and fault-tolerance, as well as ample free storage space on every host for recovering from replica failures and backfilling data during node outages. For Ceph to truly deliver fault tolerance and high availability, a realistic cluster configuration would involve at least 5 to 7 nodes and multiple storage racks. Moreover, the multitude of configurable parameters and settings in Ceph contributes to a steep learning curve, requiring a considerable trial-and-error effort to configure the system optimally for the specific cluster needs. Ceph is primarily designed for enterprise production workloads and infrastructures and may be impractical for the more average use cases.

Another notable example is GlusterFS, offering scalable, distributed network filesystems over commodity hardware, also free and open-source. GlusterFS aggregates disk storage resources from multiple remote servers into single mountable volumes. Servers are accessed over Ethernet or InfiniBand, and clients can mount a GlusterFS volume via the FUSE interface and NFS or SMB protocols.

GlusterFS provides different types of volumes that determine how files and data are routed across the backing servers. By default, files are distributed across servers without replication to save on storage space. To mitigate the risk of data loss, clients can opt for *replicated volumes* where exact copies of all files are stored on all servers. Additionally, GlusterFS offers *distributed replicated volumes* that combine both distribution and replication of files, as well as *striped* or *dispersed volumes* and *distributed dispersed volumes*.

Both the aforementioned technologies have been acquired by Red Hat and integrate

with Kubernetes through the Container Storage Interface (CSI) via their CSI driver implementations. However, it's worth noting that the Red Hat Gluster Storage, as renamed by Red Hat, is currently in its retirement phase of its lifecycle with an end of support life date of December 31, 2024. Additionally, the CSI driver for GlusterFS CSI driver is unmaintained and marked for deprecation, and the Kubernetes-integrated GlusterFS volume plugin has been removed since Kubernetes v1.26 [1].

On a final note, a recurring characteristic observed in many of today's storage systems, including high-end ones that leverage local storage, is the introduction of excessive bloat. These systems tend to incorporate numerous and intricate abstraction layers between the application and the underlying disks to implement the desired data services, such as snapshotting or replication. Abstraction layers within the critical data path inevitably lead to increased latency and a slowdown in IO performance.

1.4 Proposed Solution

As outlined so far, in this dissertation, we introduce and elucidate a simple, proof-of-concept mechanism for aggregating smaller storage resources into a single, unified storage namespace presented as persistent volumes to containerized workloads.

Our proposed solution utilizes union filesystems, specifically the MergerFS implementation, renowned for its diverse features and policy-based routing. It aggregates storage volumes from different disks and node machines over Ethernet, merging them together to provide a union mount to consumer containers. Developed exclusively for Kubernetes clusters, it functions as a volume plugin through the Container Storage Interface (CSI). We call it Union CSI.

A key aspect of its design is that Union CSI is not a standalone storage system, but rather an extension or *wrapper* for existing storage systems. In scenarios where a user's demand for storage from their preferred storage flavor cannot be fulfilled because no single disk attached to the cluster's nodes can meet the requested capacity, Union CSI can help. Union CSI splits the user's initial request for "extra-large" storage into multiple smaller and feasible requests redirected to the underlying storage system, offering

[1] https://kubernetes.io/blog/2022/09/26/storage-in-tree-to-csi-migration-status-update-1.25/#timeline-and-status

a union mount of its provided volumes. The specific requirements for the underlying storage systems are analyzed in Section 3.1. Still, Union CSI is intended to be paired with systems that efficiently utilize local storage and deployed on clusters with fast, local NVMe SSDs, enabling Union CSI to mount MergerFS over high-performance volumes.

The goal is for users to integrate Union CSI with their preferred local storage system to harness unified volumes that surpass the capacity of any available single disk in the cluster. This promises both significant scale-out and scale-up potential. Whether a disk's capacity is increased or a new node with local storage is added to the Kubernetes cluster, they can all be incorporated into the Union CSI storage pool for consumption.

1.5 Outline

From this point forward, the rest of the dissertation is structured into chapters as follows:

- **Chapter 2 Background** provides the necessary background information and context needed for the reader to proceed in the following chapters uninterrupted.
- **Chapter 3 Design** lays out the fundamental design details and architecture of the Union CSI volume plugin.
- **Chapter 4 Implementation** exposes the implementation and deployment details of Union CSI.
- **Chapter 5 Evaluation** showcases how Union CSI is used and works in action through a series of examples and provides an initial performance insight and comparison.
- **Chapter 6 Conclusion** summarizes the motivation, contents, and important points of the dissertation and outlines possible future additions.

2 Background

In this chapter, we establish context and introduce the core ideas, components, and technologies that this thesis utilizes and expands on.

2.1 Overview

We begin by introducing the fundamentals of the Kubernetes container orchestrator (CO), explaining its architecture, the workings of its declarative API, and the various abstractions it provides for running and managing applications. A particular emphasis is placed on its storage system and how it has evolved to support various types of persistent volumes for containerized workloads.

Following that, we go over the Container Storage Interface (CSI) specification for standardizing how storage is exposed to containers across different CO systems by external volume plugins. We delve into the design of a CSI-compliant volume plugin, analyzing its most prominent Remote Procedure Calls (RPCs) for provisioning and deleting, attaching and detaching, and mounting and unmounting storage volumes in a CO environment.

We elucidate how Kubernetes implements CSI, uncovering the mechanisms developed to interface with arbitrary third-party CSI volume plugins. Additionally, we provide a comprehensive example illustrating how CSI volumes are provisioned, consumed, and deleted in Kubernetes.

Lastly, we offer a concise overview of how union mounts are employed to combine dif-

ferent directory contents and filesystem instances for numerous use cases. Furthermore, we expound on the FUSE mechanism for creating and managing filesystems outside of kernel code, and we introduce the MergerFS union filesystem implementation that utilizes FUSE to aggregate storage pools, incorporating additional features compared to traditional union mounts.

2.2 Kubernetes

Kubernetes is a powerful open-source container orchestration (CO) platform that facilitates the definition and automation of containerized workloads. It handles deployment, scaling, failover, and overall management of container applications.

At its core, Kubernetes follows a declarative approach to application management. Users define the desired state of their application, and Kubernetes's internal control loops work to bring the actual state closer to the desired state, adhering to the *Operator Pattern*.

Originally developed at Google and publicly released in 2014, Kubernetes already incorporated over a decade of best practices and expertise in managing large-scale production workloads. With its widespread adoption and significant impact on the industry, Kubernetes has become one of the most popular COs and open-source projects, establishing a standard for building, deploying, scaling, automating, and maintaining cloud-native applications.

Kubernetes can run on various platforms, including bare-metal, on-premises, or on the cloud. Most major cloud providers, like Amazon (AWS), Microsoft (Azure), and Google (GCP), offer Kubernetes as a cloud-hosted managed service.

2.2.1 Architecture & Components

A Kubernetes deployment is a cluster of nodes. Nodes are machines where containers can be deployed and run. Each node has CPU, memory and storage resources that are managed by Kubernetes and exposed to cluster users. Kubernetes represents compute, network, storage, as well as custom resources with API resources and objects. Cluster administrators and users interact with these resources through the Kubernetes API.

Kubernetes clusters follow the master-slave architecture. The control plane of the master node oversees global operations, while worker nodes manage operations specific to each node. Multiple master nodes on a single cluster can co-exist for high availability, and a master node can also serve as a worker, supporting single-node clusters.

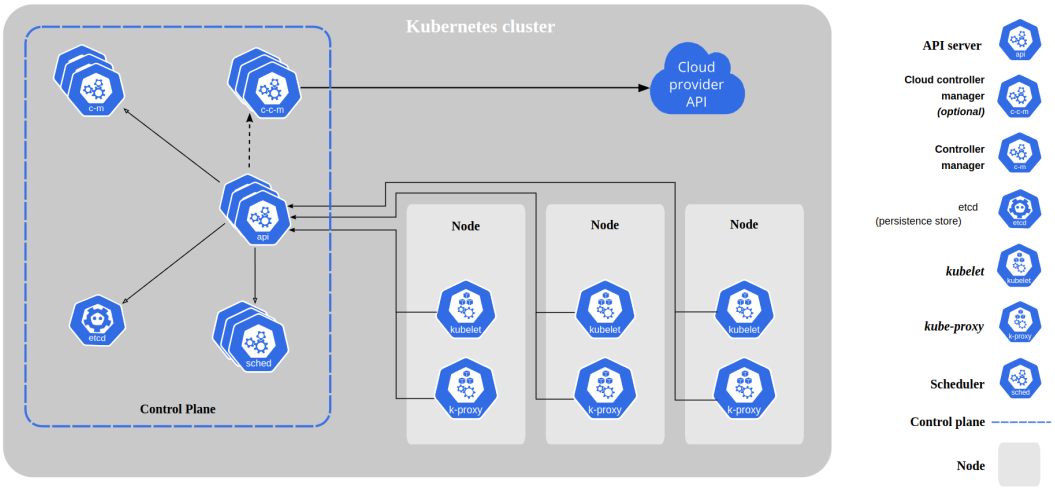


Figure 2.1: *The components of a Kubernetes cluster*

[Source: https://kubernetes.io/docs/concepts/overview/components/]

2.2.1.1 Control Plane

A Kubernetes master node hosts the control plane, the central hub of the cluster. The control plane oversees various aspects of the cluster, including the management of worker nodes and Pods, the handling of cluster-wide functions like scheduling, scaling, and resource administration, and receiving HTTP API requests and responding to cluster changes and events.

The control plane primarily comprises of five distinct components, each its own process:

`etcd`

The key-value store used by Kubernetes to persist all cluster data. etcd is an open-source, lightweight, persistent, distributed, and highly-available key-value store database. It includes built-in snapshot support, preserves previous versions, and supports watch queries. etcd often serves as the single source of truth in modern distributed systems. Kubernetes relies on etcd to store, persist, and retrieve the current and desired state of system objects at any given point in time.

In standard Kubernetes deployments, a single etcd instance is common. However, various external etcd operators exist for managing etcd clusters, offering increased availability, reduced risk of disaster and data loss scenarios, improved recovery capabilities, and simplified etcd backups and restores.

`kube-apiserver`

The API server of the control plane that exposes the Kubernetes HTTP API. It serves resource-based REST endpoints and acts as the frontend for interacting with the cluster's shared state. Each Kubernetes API resource has a corresponding access point in the API. Users and other components can configure API objects by specifying object manifests in JSON or YAML format and making REST API calls on the server. The API server validates and processes the requests, updating the state of the API objects in etcd.

`kube-apiserver` is the implementation of the API server running in a Pod on the control plane and can scale horizontally—additional instances can be added for traffic balancing.

`kube-scheduler`

The control plane component that monitors for unscheduled Pods and determines the node on which they will run based on requested and available resources. The scheduler keeps track of the total and allocated resources (CPU cores, memory size, storage capacity, etc.) on each node to understand the cluster's used and available resources. When a new Pod is created, the scheduler is responsible for selecting its destination node, considering various factors like available resources, workload requirements, user-defined constraints, policy restrictions, affinity/anti-affinity requirements, and data locality.

`kube-controller-manager`

The set of all the core Kubernetes controllers. See Section 2.2.1.4 for more information about controllers. There are various built-in Kubernetes controllers, including:

- **Node controller**: Watches for Node objects and reacts when nodes go down.
- **ServiceAccount controller**: Watches for ServiceAccounts objects and ensures a ServiceAccount named ”default” exists in every namespace.

- **Job controller**: Watches for Job objects that represent one-off batch jobs and runs Pods to completion.

Each controller operates independently but they are all compiled into the single binary of `kube-controller-manager` and run in a single process.

`cloud-controller-manager`

The set of controllers that link the Kubernetes cluster with the cloud provider. Many cloud vendors offer managed Kubernetes services in a cloud-hosted environment. `cloud-controller-manager` couples the cluster's cloud-related components with the underlying cloud platform, facilitating interfacing and management of the cloud-dependent Kubernetes resources by the cloud provider.

Similar to `kube-controller-manager`, `cloud-controller-manager` packages multiple independent controllers into a single binary. These controllers are specific to the cloud provider. If the Kubernetes cluster is on-premises or local, the `cloud-controller-manager` component is excluded from the installation.

The design of `cloud-controller-manager` utilizes a plugin system, allowing seamless integration of various cloud providers with Kubernetes.

2.2.1.2 Nodes

A worker node in a Kubernetes cluster is a machine where container workloads undergo their lifecycle as part of the Pod abstraction. The node components below are responsible for running Pods, mounting volumes on them, and forwarding network traffic.

Kubelet

The primary node agent. Kubelet registers the node with the API server and a Node object is created to represent it. When the scheduler selects a Node for a Pod to run on, Kubelet is responsible for starting, monitoring, restarting, and stopping the Pod containers, as well as mounting the specified volumes. If the Pod state is not the desired, Kubelet periodically retries to start the Pod and updates it on the API server with informative messages.

`kube-proxy`

The node network proxy. `kube-proxy` is part of the Service abstraction and is responsible for routing incoming traffic, whether from inside or outside the cluster, to Pod containers based on specified IP and port number.

Container runtime

The essential component for running and managing containers. A container runtime manages the complete lifecycle of containers, from start to termination. Kubelet interacts with container runtimes through the Container Runtime Interface (CRI) for handling containers within the Kubernetes environment. Kubernetes integrates with well-known container runtimes such as containerd, rkt, CRI-O, and virtually any CRI implementation.

2.2.1.3 Namespaces

Kubernetes utilizes namespaces to isolate resources into distinct, non-overlapping groups within a single cluster. Objects of the same resource must have unique names within a namespace but can share names across different namespaces. Kubernetes API resources are either namespace-scoped or cluster-scoped. Namespaces are themselves API objects of the Namespace resource, and when a Namespace is deleted from the cluster, all objects belonging to it are also deleted.

Namespaces prove beneficial for clusters accommodating multiple users, teams, projects, and deployments.

2.2.1.4 Controllers

A Kubernetes controller is an active reconciliation process responsible for monitoring one or more resources in the API server. It observes the objects of these resources, comparing their current state to the desired state defined in their `spec` field. If the two states are not aligned, the controller takes actions to bring the current state closer to the desired state, typically by making requests to the API server.

At the end of each iteration, whether the desired state was achieved or not, the controller updates the reconciling object's `status` field, providing the most recent observations

summarizing its current state. Clients and components rely on these status reports to understand the object's current state and take appropriate actions. This continuous process is known as a *control* or *reconciliation loop*, drawing inspiration from the field of robotics and automation.

The desired state of an object changes when its `spec` field is updated by a user. Kubernetes controllers are designed to work on the most recent desired state they encounter and may miss intermediate versions. In such cases, the system is not required to satisfy previous states before moving to the next one; it will pick up from the most recently observed state. Thus, Kubernetes exhibits a *level-based* behavior rather than an *edge-based* one.

A common pattern is for a controller to be dedicated to one resource type for following its desired state, and managing another resource by creating, updating and deleting instances of it, to achieve the desired state. For instance, the built-in ReplicaSet controller monitors ReplicaSet objects and ensures the specified number of Pods are always present and operational. It scales up Pods when their desired number is higher, creating new copies from the ReplicaSet template, and scales down Pods when there is an excess by selecting and deleting surplus Pods. Another example is the DaemonSet controller, which ensures a specified number of replica Pods are running on every node (or specified subset of nodes), populating newly added nodes with Pods and deleting Pods from removed nodes.

Label selectors can influence the sets of objects a controller manages. For example, the ReplicaSet controller attempts to take ownership of Pods with labels matching the ReplicaSet selector when scaling up and releases owned Pods that no longer match when scaling down.

The core Kubernetes controllers for the built-in resources (Deployment, Job, PersistentVolume, etc.) are packaged in the `kube-controller-manager` Pod of the control plane. Additionally, custom resources managed by custom controllers can be designed and deployed to extend the Kubernetes API in various ways (see Section 2.2.2.2).

Of particular interest and importance to the Kubernetes storage layer are the PersistentVolume and AttachDetach controllers of `kube-controller-manager`.

PersistentVolume controller

The PersistentVolume controller watches for PersistentVolume (PV) and PersistentVolumeClaim (PVC) objects in the API server. It invokes the provisioning and deletion of volumes (directly or indirectly), creating and deleting PVs, and binding together PVCs and PVs.

In the case of dynamic provisioning, the controller detects an unbound PVC requesting a volume of a particular StorageClass. It then updates the PVC with the name of the volume plugin that is responsible for provisioning the volume by adding the `volume.kubernetes.io/storage-provisioner` annotation with the `provisioner` value of the StorageClass.

If the controller recognizes the plugin's name as one of the in-tree volume plugins (volume plugins that are part of the Kubernetes code), it invokes the `Provision` method of the specified plugin to create a new volume. Upon successful creation, the controller generates a PV to represent the newly provisioned volume and binds the PV to the PVC with a `spec.claimRef` object reference.

If the requested plugin is not recognized, considered an out-of-tree volume plugin (an external volume plugin installed by an administrator, like a CSI plugin), the controller broadcasts an Event on the PVC, indicating it is waiting for the external plugin to do its job, and does nothing further. The external provisioner, deployed with the external plugin and functioning as a PV/PVC controller similar to the built-in controller, detects that a PVC refers to its registered plugin with the `volume.kubernetes.io/storage-provisioner` annotation and triggers the plugin's provision operation. For CSI plugins, this provision operation corresponds to the `CreateVolume` RPC. The external provisioner then creates a PV instance and binds it to the PVC, similar to the internal PersistentVolume controller. An example of an external provisioner is given in Section 2.4.2, regarding the `external-provisioner` developed by the Kubernetes team as a sidecar container for external CSI plugins.

In all cases, when the PersistentVolume controller detects a PV bound to a PVC, it proceeds to bind the PVC to the PV, setting the PVC's `spec.volumeName` to the PV's `name`, completing their bidirectional binding.

Similarly, when a user deletes the PVC, the PersistentVolume controller takes either of two actions: it either calls the `Delete` method of the in-tree volume plugin to delete the

volume or defers to the external provisioner to trigger the delete operation of the registered plugin. For CSI plugins, this delete operation corresponds to the `DeleteVolume` RPC. Upon successful deletion, the controller associated with the plugin responsible for the volume deletion (either the PersistentVolume controller or the external provisioner) proceeds to delete the released PV from the API.

AttachDetach controller

The AttachDetach controller watches for Pods utilizing volume plugins that are scheduled on Nodes and invokes the attachment and detachment of volumes. For additional insights on how the AttachDetach controller initiates the attach/detach operations of CSI volume plugins, refer to Section 2.4.1.

2.2.1.5 Workloads

Kubernetes provides various abstractions for workloads, starting with the smallest unit—the Pod. Higher-level abstractions manage sets of Pods in different ways. Users can select and configure the workload resources that best suit their application needs, and the resource's controller will strive to achieve the specified state.

Pod

The basic deployable unit in Kubernetes, a Pod comprises one or more containers colocated on the same node. All containers within a Pod share namespaces, storage, and network resources. These containers are intended to be parts of the same application instance, benefiting from shared resources, dependencies, and communication.

The Pod API resource includes `spec` and `status` fields. The scheduler assigns Pods to nodes, updating the Pod's `spec.nodeName` field accordingly. Once a node is chosen, the Kubelet on that node initiates the creation and preparation of the specified containers. If the Pod starts successfully, it remains running on the node until termination.

Pods do not self-heal autonomously. In the event of a node failure, all Pods on that node fail and are eventually removed from the API. Each Pod must be recreated, either manually or by the managing controller.

Pods adhere to a specific lifecycle. The PodStatus object of the `status` field is a large

nested structure that provides real-time information about the Pod's volumes, assigned IPs, and containers. The Pod's phase (`status.phase` field) summarizes its lifecycle stage. Every Pod starts from the `Pending` phase, progresses to `Running` if at least one of its main containers starts successfully, and finally transitions to either the `Succeeded` or `Failed` phases based on whether any container in the Pod terminated in failure. Similarly, Kubernetes monitors and reports the lifecycle stage of each individual container within the Pod. Container states include `Waiting`, `Running`, and `Terminated`.

When a Pod object is being deleted, it is first given a window for its processes to gracefully terminate before they are forcefully shutdown. Specifically, when a Pod is requested for deletion, Kubernetes updates the Pod in the API server to show up as "Terminating" and a grace period is triggered (default is 30 seconds). On the node where the Pod was running, Kubelet detects that the terminating Pod has a grace period set. Kubelet then proceeds to ask the container runtime to stop the containers of the Pod by sending the `SIGTERM` signal to the PID 1 process within each container. Processes listening for the `SIGTERM` signal then have an opportunity to perform any necessary cleanup operations. Once the grace period expires, the `SIGKILL` signal is sent to any remaining processes and the Pod is removed from the API server.

Users typically create and manage Pods indirectly through higher-level resources like ReplicaSets or Deployments.

ReplicaSet

A ReplicaSet is responsible for maintaining a specified number of Pods at all times. The ReplicaSet spec field most notably includes:

- **`selector`**: Label selector query for identifying matching existing Pods to adopt.
- **`replicas`**: Desired number of replica Pods the ReplicaSet controller tries to maintain.
- **`template`**: Pod template (`metadata` and `spec` fields) used for generating new Pods when their current number is insufficient.

The ReplicaSet controller scales up or down Pods in order to align with the desired replica count. During scaling up, whether acquiring an existing Pod through a successful selector query or creating a new replica Pod, the controller sets on the Pod an one-

way link to its owner ReplicaSet object through a `metadata.ownerReferences` pointer. This reference is used by the ReplicaSet controller to track the existing Pods owned by each ReplicaSet instance.

Deployment

Operating in a manner similar to a ReplicaSet by specifying a Pod template and overseeing sets of Pods, a Deployment manages Pods through a ReplicaSet instance. It is a higher-level construct than a ReplicaSet, with the Deployment controller providing additional functionalities over its underlying Pods, such as rolling updates, rollbacks, and versioning of the application. The Deployment is the recommended workload resource for maintaining a stable group of stateless Pods.

StatefulSet

A StatefulSet manages the deployment and scaling of stateful applications. Unlike traditional Deployments, a StatefulSet assigns a unique and persistent identity to each Pod it creates, making it suitable for applications that require stable network identifiers or persistent storage.

DaemonSet

A DaemonSet guarantees a Pod is running on every Node in the cluster or a designated subset. When new nodes join the cluster, corresponding Pods are automatically deployed onto them. Conversely, when nodes leave the cluster, those Pods are removed. The DaemonSet resource is useful in scenarios where achieving node coverage is important, such as node monitoring, log collection and storage services.

The above list of workload resources is not exhaustive.

2.2.1.6 Storage

Container filesystems are ephemeral by default. When a running container stops or crashes and is later restarted, all prior data changes are permanently lost. Nevertheless, stateful applications, such as databases, demand data persistence across application crashes, restarts, and reschedules. To address these challenges, Kubernetes has de-

vised mechanisms and abstractions, allowing Pod containers to request and access both ephemeral and persistent volumes that cater to their workload's specific requirements.

Volumes

Kubernetes supports various types of file or block storage volumes, whether ephemeral or persistent, local or remote. A Pod can define and utilize any number of volumes for its containers. Ephemeral volumes undergo the same lifecycle as the Pod and are destroyed when the Pod is deleted. However, persistent volumes have a lifecycle independent of a Pod, and their contents persist through restarts.

At its core, a Kubernetes volume is a directory on a node that Kubelet mounts on Pod containers. The volume can be either a formatted filesystem or a raw block device, and the underlying medium can be either locally- or network-attached storage disk. Those specifics, along with volume performance characteristics and supported features, are determined by the storage provisioner of the volume.

A Pod can utilize any number of volumes for its containers. The volumes used are specified in the `spec.volumes` list, and the target container along with its corresponding mount point are specified in `spec.containers[*].volumeMounts`. A running Pod container process will view and access the original image root filesystem, along with any externally mounted filesystems under the mount points in `spec.containers[*].volumeMounts`.

Some of the many Kubernetes volume types that can be specified in a Pod's `spec.volumes` are listed below:

- **`emptyDir`**: An initially empty directory that all containers in a Pod can read and write to. Data in `emptyDir` survive container crashes and restarts. However, if the Pod is removed from the node, the `emptyDir` volume is wiped out.
- **`configMap`**: A read-only directory containing configuration data for the Pod to consume.
- **`hostPath`**: A directory mounted from the host node's filesystem on the Pod container. `hostPath` volumes pose many security risks and are intended for privileged applications that require access to the host filesystem, such as Docker's `/var/lib/docker` or Kubelet's `/var/lib/kubelet`, or for clusters on local computers.

- **`local`**: A local to the node storage device backed by a disk, partition or directory. By default, `local` PersistentVolumes can only be pre-provisioned and do not support dynamic provisioning. The PV must have node affinity set (`spec.nodeAffinity` field), so Pods using the local PV can only be scheduled on the node where the volume is mounted.
- **`fc`**: An existing Fibre Channel (FC) block storage volume.
- **`iscsi`**: An existing iSCSI (SCSI over IP) volume. The volume can be pre-populated and its data is not affected by the deletion of the Pod. `iscsi` volumes can be mounted as read-only by many consumer containers but can only be used in read-write by a single consumer. An iSCSI server and the `iscsi` volume must already exist to be used by a client Pod.
- **`nfs`**: An existing NFS (Network File System) volume. Similar to `iscsi`, but `nfs` volumes can be mounted by multiple writers simultaneously.
- **`csi`**: A ephemeral volume provisioned by an arbitrary storage system through the Container Storage Interface (CSI).
- **`persistentVolumeClaim`**: A reference to a persistent volume through a PersistentVolumeClaim.

Static & dynamic provisioning

Cluster administrators have the ability to create new volumes from specific storage systems, along with PersistentVolume objects that represent these volumes in the Kubernetes cluster. Users can then make use of these pre-created volumes in their Pods through a PersistentVolumeClaim that references the PersistentVolume. This storage provisioning method is known as static provisioning, where users can only choose from and consume existing volumes.

Conversely, dynamic provisioning occurs when a new volume is created by the user on-the-fly. Dynamically provisioned volumes offer convenience, as they are automatically provisioned by a volume plugin in the cluster, and users can specify desired characteristics such as storage size and access modes. Dynamic provisioning starts with the StorageClass API.

StorageClass

Storage and cloud providers utilize StorageClass objects to represent and advertise the

different ”classes” or ”tiers” of storage they offer. The StorageClass plays a crucial role in the dynamic provisioning process in Kubernetes. Instead of pre-creating volumes, administrators create a set of StorageClasses in the cluster. Users can then choose from these classes and create volumes by specifying the StorageClass name in a PersistentVolumeClaim.

If a PVC does not specify a StorageClass in `spec.storageClassName`, the default StorageClass will be used. There can be only one default StorageClass in a single cluster. PVCs that specify the empty StorageClass, **””**, effectively disable dynamic provisioning for themselves and can only be bound to PVs that do not belong to a StorageClass.

The StorageClass API is not namespaced and does not follow the standard `spec/status` format. Its key fields include:

- **`provisioner`**: The specific volume plugin responsible for provisioning, managing, and deleting volumes of this StorageClass.
- **`parameters`**: Key-value map that describes the characteristics of volumes belonging to the StorageClass, such as maximum IOPs, filesystem type, encryption used, etc. The parameter keys are understood and accepted only by the associated provisioner and are opaque to Kubernetes.
- **`volumeBindingMode`**: Controls when the dynamic provisioning of a volume should occur: when a volume is first requested through a PersistentVolumeClaim or when a Pod using the volume via the PersistentVolumeClaim is scheduled on a node. The `Immediate` value expresses the former, while the `WaitForFirstConsumer` the latter. Provisioners and storage backends that provide volumes globally accessible on any node in the cluster, like NFS or remotely attached volumes, can work with StorageClasses of `Immediate` volume binding mode. However, many storage backends require knowledge of the destination node to allocate the specified volume locally. Administrators must use the `WaitForFirstConsumer` volume binding mode for such backends to delay the provisioning process until a Pod is scheduled and the node is known. Otherwise, it may result in unschedulable Pods.

PersistentVolume

The PersistentVolume resource, sometimes simply referred to as PV or *volume*, repre-

sents a statically or dynamically provisioned volume in the Kubernetes cluster. PVs are like Kubernetes volumes defined in a Pod but have a lifecycle independent of any Pod that uses them. The PV resource is not namespaced.

A user can claim a PersistentVolume by creating a PersistentVolumeClaim that refers to a StorageClass with a specific amount of storage requested and certain access modes. The PersistentVolume controller of `kube-controller-manager` detects the unbound PVC and attempts to find an existing, unbound, matching PV. If such a PV does not exist, the controller invokes the associated provisioner specified in the StorageClass to provision a new volume. When the requested volume is provisioned, the controller creates a new PV object in the API server to represent the volume. The PVC and PV instances are bound to each other by the controller through a bi-directional binding, meaning the volume belongs only to the user now who can manage it through the PVC for as long as desired.

A Pod can use a PV through a PVC by including a `persistentVolumeClaim` section with the PVC's name in the Pod's `spec.volumes` list. The Pod and its used PVCs must all exist in the same Kubernetes namespace. When the Pod is scheduled, the volume backing the PV bound to the PVC will be accessible by the Pod's container(s). A PV bound to a PVC and a PVC currently in use by a Pod are protected by finalizers and cannot be removed from the API server when requested for deletion. A bound PVC that is not used by a Pod can be deleted, and the bound PV is released and can be deleted or reclaimed, depending on its reclaim policy.

The PV `spec` has the following fields:

- **`capacity`**: The requested lower and upper storage size bounds.
- **`persistentVolumeSource`**: The details regarding the actual storage asset backing the PV. The currently supported (Kubernetes v1.28) volume plugins are: `hostPath`, `local`, `fc`, `iscsi`, `nfs`, and `csi`. Vendor-specific volume plugins, like AWS EBS, Azure Disk, CephFS, etc., were part of the core Kubernetes source code and binaries but have started becoming deprecated with the introduction of CSI and are no longer provided by default Kubernetes installations. Instead, third-party plugins have to be deployed from their respective vendors in order to utilize them. Only one volume plugin can be specified at a time in the spec of the PV. This field is inlined, meaning the `persistentVolumeSource` key name is omitted and its

value is promoted in the `spec` field in the PV returned from the API server.

- **`accessModes`**: The possible access modes requested for the volume, including `ReadWriteOnce` (`RWO`), `ReadWriteMany` (`RWX`), `ReadOnlyMany` (`ROX`) and `ReadWriteOncePod` (`RWOP`).
- **`claimRef`**: The PV's end of the bi-directional PVC-PV binding.
- **`persistentVolumeReclaimPolicy`**: The policy to follow when the PV is released from a PVC. `Retain` means the PV and its underlying volume are preserved in the cluster for manual reclamation by an administrator, `Recycle` means the PV and its volume are preserved in the cluster but the data are whipped clean, and `Delete` means the PV and its volume are permanently deleted. `Recycle` is currently deprecated.
- **`storageClassName`**: The name of the StorageClass that the volume belongs to; an empty value "" means that the volume does not belong to any StorageClass.
- **`mountOptions`**: The mount options to use when the volume is mounted on a node.
- **`volumeMode`**: Specifies whether the volume appears inside a container as a formatted filesystem or a raw block device, dictated by the `Filesystem` and `Block` values, respectively.
- **`nodeAffinity`**: The set of node constraints defining what nodes the volume is accessible from. `nodeAffinity` is a label selector query that must match the labels of a Node object, influencing the scheduling of Pods that use the volume.

The PV's phase (`status.phase` field) can be one of the following:

- **`Available`**: The PV is not yet bound to a PVC.
- **`Bound`**: The PV is bound to a PVC.
- **`Released`**: The previously bound PVC is deleted and the volume is not yet reclaimed by the cluster.
- **`Failed`**: The released PV failed to get reclaimed or deleted.

PersistentVolumeClaim

The PersistentVolumeClaim resource, sometimes simply referred to as PVC or *claim*, is a request for a volume with specific characteristics. A PVC becomes bound to an existing or newly provisioned PV that matches the requested volume specifications. The PVC resource is namespaced, in contrast to the PV resource which is cluster-scoped.

Some of the PVC `spec` fields are:

- **`accessModes`**: The same access modes as in the PV `spec`.
- **`volumeMode`**: The same volume mode (filesystem or block) as in the PV `spec`.
- **`resources`**: The requested storage size.
- **`storageClassName`**: The requested StorageClass that the volume must belong to.
- **`volumeName`**: The name of the bound PV. The PVC's end of the bi-directional PVC-PV binding.
- **`selector`**: Label selector query to consider for PVs for binding.

The PVC's phase (`status.phase` field) can be one of the following:

- **`Pending`**: The PVC is not yet bound to a PV.
- **`Bound`**: The PVC is bound to a PV.
- **`Lost`**: The PVC was bound to a PV but that PV no longer exists.

2.2.2 API & Resources

The access point to Kubernetes is the API server of the control plane (`kube-apiserver`), which exposes an HTTP resource-based declarative API. Clients and external components query and manipulate the state of API objects through REST calls made on the API server. The state of all cluster objects is persisted using the etcd control plane component.

In addition to the standard HTTP verbs—`GET`, `POST`, `PUT`, `PATCH`, and `DELETE`—Kubernetes uses its own custom verbs, often written in lowercase to distinguish them from the HTTP verbs. For example, the `get` request returns only one resource instance, while the `list` request returns a collection of instances that match the request's criteria. A `watch` request can be paired with `get` or `list` to track changes on resources. Moreover, the traditional `PUT` verb is distinguished between `create` or `update` in Kubernetes, based on whether the instance already exists or not. However, an `update` is not a `PATCH`; `PATCH` is `patch`.

Collections of resource objects are published under the `/api` and `/apis` endpoints. Objects belonging to the same group are found at `/apis/<group>`. Each group can have

different versions, indicating different levels of stability and support, resulting in different `/apis/<group>/<version>` paths. For example, the Deployment resource belongs to the `apps` group, which currently (Kubernetes v1.28) has the `v1`, `v1beta1`, and `v1beta2` versions. Objects of multi-version groups can be equally viewed and updated from each one of their versions, with the API server handling the conversion. This is important for preserving objects when one of their versions is deprecated and removed.

As a result, API resources are defined by their API group, resource type (e.g., Pod, Deployment, Service), namespace (for namespaced resources), and name. For cluster-scoped resources, the endpoint is `/apis/<group>/<version>/<type>/<name>`, and for namespaced resources, it is `/apis/<group>/<version>/namespaces/<namespace>/<type>/<name>`. For the `core` group, the group name is omitted in the endpoint and `apiVersion` field of objects.

2.2.2.1 API Objects

Kubernetes API objects are persistent data structures. These objects typically represent concrete entities or abstract concepts within the cluster, such as node machines, groups of running containers, network resources, storage resources, namespaces, etc. A Kubernetes object definition is a specification of the object's desired state. Once the object is created in the API server, Kubernetes continually works to achieve its desired state.

The following top-level fields are required in the object's definition file:

- **`apiVersion`**: The version of the created object in the Kubernetes API.
- **`kind`**: The kind or *schema* of the object (e.g., Pod, Deployment, PersistentVolume).

Next, the `metadata` nested object fields include both required and optional fields that help uniquely identify the object instance. Some of the `metadata` fields are:

- **`name`**: The unique name of the object within its current namespace.
- **`namespace`**: The namespace to which the object belongs. If omitted, the `default` namespace is used.

- **`uid`**: A system generated, unique in time and space identifier that distinguishes objects of the same name (and namespace if namespaced) between previous and future recreations.
- **`labels`**: A map of key-value pairs intended for organizational and grouping purposes by end users.
- **`annotations`**: A map of key-value pairs intended for attaching metadata by third-party automations and external tooling.
- **`finalizers`**: A list of strings that blocks the deletion of the object. When an object is deleted, Kubernetes does not remove the object if it has a non-empty `finalizers` list. Instead, Kubernetes adds a `deletionTimestamp` to the object, transitioning it to a terminating state. New finalizer entries cannot be added in this state. A controller responsible for the object detects the `deletionTimestamp`, performs any necessary clean-up operations, and removes one or more finalizers that it is responsible for. When the `finalizers` list becomes empty, the object is removed from the API.
- **`creationTimestamp`**: A system generated string representing the date and time the object was created.
- **`deletionTimestamp`**: A system-generated string representing the date and time at which the object will be deleted from the API, once the `finalizers` list is empty.
- **`ownerReferences`**: A list of objects dependent by this object, influencing its garbage collection. Only one entry can be the *controller*; the owner object responsible for this object.

If the resource is not solely for persisting and retrieving data (a StorageClass is a good example), then it will most probably have the standard top-level `spec` and `status` fields:

- **`spec`**: The desired state of the object, as specified by the user. The `spec` can have multiple flat or nested fields, both mutable and immutable, and varies for different resources. The convention is that `spec` is user-readable/writeable and controller-readable.
- **`status`**: The current state of the object, as observed by the relevant controller. The `status` usually follows specific standards such as including a `conditions`

array field or a `phase` field. The convention is that `status` is user-readable and controller-readable/writeable.

Objects that contain both `spec` and `status` should not include other top-level fields other than the standard `apiVersion`, `kind`, and `metadata` fields.

Kubernetes objects can be expressed in JSON or YAML format, in files known as *manifests*. The manifest is used to create the object in the Kubernetes API server, either through a client tool or by making a direct REST call. The object definition is serialized in the HTTP body in JSON.

Listing 2.1 shows an example of a Deployment manifest in YAML.

Listing 2.1: *Example of a Deployment*

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

2.2.2.2 Custom Resources & Controllers

The Kubernetes API is extensible, allowing users to define custom resources (abbreviated as CRs) that are not part of the standard Kubernetes installation. Custom resources are declared through CustomResourceDefinitions (CRDs), a special resource whose objects dynamically register and remove custom resources on the cluster without modifying the Kubernetes source code. When a CRD object defining a new resource is created, a new API endpoint is allocated for that resource to access its objects, similar to a built-in resource. Deleting the CRD removes all custom resource objects, the endpoint disappears, and Kubernetes no longer recognizes the custom resource.

Many new Kubernetes resources and functionalities were introduced through CRDs, and some have been promoted to built-in resources over time.

For new dynamic use cases, custom resources can be paired with custom controllers that are not part of `kube-controller-manager`. This combination provides a declarative API, where users define the desired state of a custom resource object, and the controller reconciles it to achieve its desired state, updating it with the latest observations.

The Kubernetes *Operator Pattern* is based on custom resources and controllers. An *operator* is a controller that encapsulates the role of a human operator responsible for a service or sets of services. Operators are employed to look after specific resources and applications, responding to updates and automating their behavior.

2.2.2.3 API Clients

The Kubernetes API can be accessed from the command-line with tools like `kubectl` and `kubeadm`, using official client libraries in various programming languages when developing Kubernetes applications, or directly with REST calls.

2.3 Container Storage Interface (CSI)

The Container Storage Interface (CSI) is an industry standard for exposing arbitrary block and file storage storage systems to containerized workloads on Container Orchestrator systems (COs). CSI defines both how third-party storage providers can expose their storage solutions in any CO environment that implements CSI, and how COs can interface with external CSI volume plugins. The adoption of CSI decouples the storage layer of COs from external storage systems, and enables both to develop and release features at their own pace, enhancing flexibility, stability, and security. More details about CSI can be found in the Container Storage Interface `spec` GitHub repository [1], which includes the CSI specification and library.

CSI is actively developed and enhanced with new RPCs, fields, and features in new releases. The overview provided in this thesis involves CSI v1.8.0 [2].

[1] https://github.com/container-storage-interface/spec/blob/master/spec.md
[2] https://github.com/container-storage-interface/spec/tree/v1.8.0

2.3.1 CSI Remote Procedure Calls (RPCs)

The CSI API is based on Remote Procedure Calls (RPCs), using the gRPC framework developed by Google. These RPCs invoke operations such as:

- Dynamic provisioning and deletion of a volume.
- Attaching and detaching a volume on a node.
- Mounting and unmounting a volume on a node.
- Expanding an existing volume's size.
- Creating and deleting a snapshot from an existing volume.
- Creating a new volume from a snapshot.

The CSI specification focuses on the RPCs a CSI-compliant volume plugin must implement and serve to integrate with any number of different COs and expose its storage. The component of a volume plugin that implements the CSI RPCs is called a CSI driver. COs (the gRPC client) communicate with CSI drivers (the gPRC server) through RPCs via a gRPC endpoint, which is a UNIX domain socket shared between the CO's CSI components and the driver.

The CSI RPCs are divided into three distinct sets:

- **Identity service**: RPCs that return identity and readiness information about the plugin. The CO uses this information to uniquely identify the plugin and understand its general capabilities. The Identity service is required; every plugin must implement its three RPCs:
  - **`GetPluginInfo`**: Returns the name and vendor version of the plugin. Two plugin instances belong to the same plugin deployment if they report the same name and version in the `GetPluginInfoResponse`.
  - **`GetPluginCapabilities`**: Returns capabilities that describe all plugin instances.
  - **`Probe`**: Returns information about the plugin's instance initialization status and whether it is ready to accept Controller and Node service RPCs.
- **Controller service**: RPCs that involve operations that can run from any node in the cluster (e.g., provisioning, attaching).

- **Node service**: RPCs that involve operations that must run on the node whereupon the plugin's volume will be published (mounted) on.

Each CSI RPC receives a request structure and returns a response structure and a standard gRPC status structure for error handling. The request, response, and status are serialized through the CSI endpoint using Protocol Buffers (Protobuf). Each request and response structure of each RPC has a set of fields that are thoroughly documented, including required fields, optional fields, relationships between fields, default values, size limits, etc.

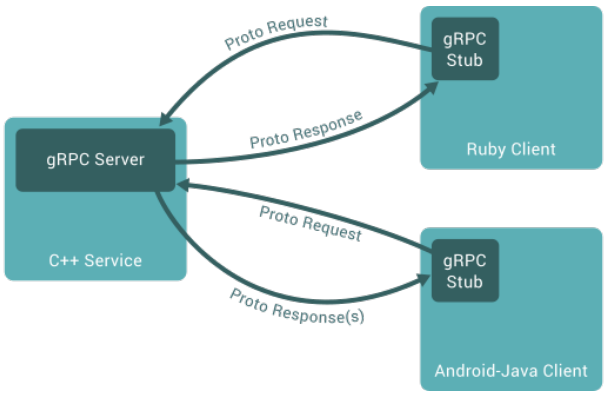


Figure 2.2: *gRPC overview*

[Source: https://grpc.io/docs/what-is-grpc/introduction/#overview]

RPCs can time out and be retried by the CO later on. The CSI plugin must ensure all mutating RPCs are idempotent. An RPC that changes the state of the system, made with the same parameters that constitute the *ID* of the call, must succeed only once. The exact idempotency requirements of each RPC are documented in the CSI specification.

The CSI RPCs that drive a volume through its fundamental lifecycle from creation to deletion in a CO environment are described below:

`CreateVolume` (Controller service): The CO invokes this RPC for the plugin to provision a new volume on behalf of a user. The volume's requested size, type (file or block), access modes, volume source, specific parameters, and topology constraints are all specified in the `CreateVolumeRequest`. The CO also passes a proposed unique identifier for the requested volume that the plugin can choose to use as a token to refer to the volume and drive the idempotency process. For idempotency, if a volume corresponding to

this identifier already exists and is compatible with the parameters passed in the `CreateVolumeRequest`, then the plugin must do nothing more and return an OK error code in the `CreateVolumeResponse`. On success, the plugin returns in the `CreateVolumeResponse` the volume identifier it shall use from now on to refer to the provisioned volume so the CO can store it and pass it in subsequent RPCs.

`DeleteVolume` (Controller service): The CO invokes this RPC for the plugin to delete a volume. This RPC is the negation of `CreateVolume`. The volume identifier returned by the plugin in `CreateVolume` is passed in the `DeleteVolumeRequest`. For idempotency, if a volume of this identifier does not exist, then the plugin must do nothing more and return an OK error code in the `DeleteVolumeResponse`.

`ControllerPublishVolume` (Controller service): The CO invokes this RPC for the plugin to make a volume available on a node. This process is considered the attachment of a volume on a node and the plugin must not assume that the RPC is executed on the destination node. The volume and node identifiers are passed in the `ControllerPublishVolumeRequest` along with additional volume parameters. For idempotency, if the volume of the given volume identifier is already published on the node of the given node identifier and is compatible with the given parameters, then the plugin must do nothing more and return an OK error code in the `ControllerPublishVolumeResponse`.

`ControllerUnpublishVolume` (Controller service): The CO invokes this RPC for the plugin to perform the necessary operations for making a volume available on a different node. This process is considered the detachment of a volume from a node and the plugin must not assume that the RPC is executed on the node that the volume is published on. This RPC is the negation of `ControllerPublishVolume`. The volume and node identifiers are passed in the `ControllerPublishVolumeRequest`. For idempotency, if the volume of the given volume identifier is not published on the node of the given node identifier, or if the volume or node cannot be found and the volume is safely considered detached from the node, then the plugin must do nothing more and return an OK error code in the `ControllerUnpublishVolumeResponse`. If the plugin implements the optional `NodeStageVolume`/`NodeUnstageVolume` RPCs, then the CO must ensure that this RPC is called after all `NodeUnpublishVolume` and `NodeUnstageVolume` calls on the

volume are successful.

`NodeStageVolume` (Node service): The CO invokes this RPC for the plugin to prepare a volume published on a node to be consumed by a workload by `NodePublishVolume`. The plugin can assume that the RPC is executed on the same node that the volume is published on. This RPC is optional for plugins that implement the Node service and it is called when a workload is scheduled on a node for the first time. The CO passes in the `NodeStageVolumeRequest` the volume identifier, the staging directory that the plugin should mount the volume on, and additional volume parameters. This RPC is useful for plugins that need to perform first-time operations to prepare a volume, like formatting and partitioning a disk or creating auxiliary files and directories. For idempotency, if the volume of the given volume identifier is already staged at the given staging directory and compatible with the given parameters, then the plugin must do nothing more and return an OK error code in the `NodeStageVolumeResponse`. If the plugin implements this RPC and `ControllerPublishVolume`, then the CO must ensure that, for the given volume and node, `ControllerPublishVolume` is successfully called before `NodeStageVolume`, and `NodeStageVolume` is successfully called before `NodePublishVolume`. Additionally, the CO must ensure that this RPC is successfully called once per volume per node.

`NodeUnstageVolume` (Node service): The CO invokes this RPC for the plugin to unstage a volume from a previously staged directory. This RPC is the negation of `NodeStageVolume`. The plugin can assume that the RPC is executed on the same node that the volume is published on. This RPC is optional for plugins that implement the Node service and it is called when a workload has finished with the volume on the node and/or the volume is to be moved on a different node. The CO passes in the `NodeUnstageVolumeRequest` the volume identifier and the staging directory that the plugin must unstage the volume from. For idempotency, if the volume of the given volume identifier is already unstaged from the given staging directory, then the plugin must do nothing more and return an OK error code in the `NodeUnstageVolumeResponse`. If the plugin implements this RPC and `ControllerUnpublishVolume`, then the CO must ensure that, for the given volume and node, `NodeUnpublish` is successfully called before `NodeUnstageVolume`, and `NodeUnstageVolume` is successfully called before `ControllerUnpublishVolume`.

`NodePublishVolume` (Node service): The CO invokes this RPC for the plugin to mount a volume on a specified path to be consumed by a workload. The plugin can assume that the RPC is executed on the same node that the volume is published on. The CO passes in the `NodePublishVolumeRequest` the volume identifier, the staging directory of `NodeStageVolume`, the target directory that the plugin should mount the volume on, and additional volume parameters. If the volume was previously staged in `NodeStageVolume`, then the plugin should mount (bind) the filesystem from the staging to the target path. On success, the CO will further mount the target path within the workloads container filesystem. For idempotency, if the volume of the given volume identifier is already mounted on the given target directory and compatible with the given parameters, then the plugin must do nothing more and return an OK error code in the `NodePublishVolumeResponse`. If the plugin implements this RPC and `ControllerPublishVolume`, then the CO must ensure that, for the given volume and node, this RPC is called after `ControllerPublishVolume` is successfully called. Additionally, depending on the capabilities of the plugin, this RPC can be successfully called multiple times for the same volume and different target directories at the same time (multi-consumer volumes).

`NodeUnpublishVolume` (Node service): The CO invokes this RPC for the plugin to unmount a volume from a specified path. This RPC is the negation of `NodePublishVolume`. The plugin can assume that the RPC is executed on the same node that the volume is published on. This RPC shall be called for the same volume at least once for each target path that was setup in a previous `NodePublishVolume`. The CO passes in the `NodeUnpublishVolumeRequest` the volume identifier and target directory that the plugin must unmount the volume from. For idempotency, if the volume of the given volume identifier is already unmounted from the given target directory, then the plugin must do nothing more and return an OK error code in the `NodeUnpublishVolumeResponse`. If the plugin implements this RPC and `ControllerUnpublishVolume`, then the CO must ensure that, for the given volume and node, this RPC is successfully called before `ControllerUnpublishVolume` is called.

The above list of RPCs is not exhaustive. The complete CSI RPC API can be found and inspected in Protobuf or Go form in the GitHub `container-storage-interface/spec` repository.

A volume's lifecycle steps, aligned with the previously mentioned RPCs and their relative order imposed by the CO, and if the plugin supports the optional `NodeStageVolume`/`NodeUnstageVolume` RPCs, are illustrated in Figure 2.3. If the plugin does not need and support the staging phase, then it is simply omitted, and `NodePublishVolume` succeeds `ControllerPublishVolume`.

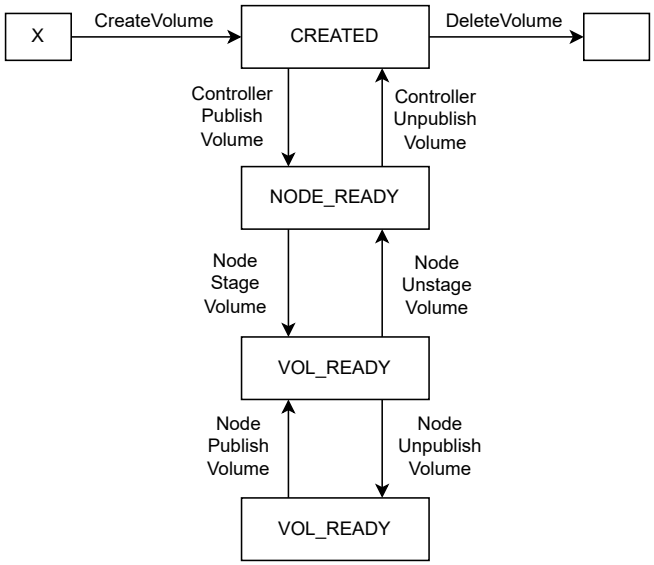


Figure 2.3: *The lifecycle of a dynamically provisioned volume in CSI, in case the plugin supports staging and unstaging volumes*

2.3.2 CSI Plugin Architecture

As mentioned, all RPCs of the Identity service are required by all instances of a CSI plugin. The RPCs of the Controller and Node services are optional. A plugin instance can choose to implement and service some or all of the Controller or Node service RPCs, which is communicated by advertising the associated capabilities. A CSI plugin running on a node can be considered one of the following:

- ***Node Plugin***: A gRPC endpoint serving the Node service RPCs. This plugin required direct access and privileged rights on the host filesystem.
- ***Controller Plugin***: A gRPC endpoint service the Controller service RPCs. This plugin usually does not require direct access and privileged rights on the host filesystem.
- A single gRPC endpoint serving both the Controller and Node service RPCs, sometimes called a *unified Plugin*.

A minimally viable yet fully functional CSI plugin deployment must consist of both *Node* and *Controller Plugins*. At least one *Controller Plugin* is required to drive operations such as creating, attaching, snapshotting, cloning, resizing, and querying volumes. At least one *Node Plugin* is required on every node where a volume is to be published. Cluster administrators can choose whether to deploy more than one *Controller Plugin* or *Node Plugin* per node for high availability, as well as explore different deployment combinations and architectures, such as unified or *split-only Plugins*, separate *Controller-only* and *Node-only Plugins* on the same node, *Controller-only Plugins* on the master node only, etc.

2.4 Kubernetes & CSI

Prior to the Container Storage Interface (CSI), volume plugins in Kubernetes were ”in-tree”, meaning they were part of the Kubernetes source code and were compiled, built, and shipped with the core Kubernetes binaries. This design approach posed several challenges:

- Storage vendors had to inspect and understand Kubernetes code and comply with the Kubernetes release process for maintenance and bug fixes.
- Storage vendors had to make their plugin's source code public.
- Kubernetes binaries had to satisfy the plugin's dependencies instead of the plugin being a separate container with self-managed dependencies.
- Bugs and security issues in a plugin affected Kubernetes components.

CSI became Generally Available (GA) in Kubernetes v1.13 (2018). With the adoption of CSI, Kubernetes and third-party volume plugin code and logic are decoupled. Storage providers can develop a CSI-compliant plugin once and deploy it across different COs. Kubernetes can discover, register, and interact with arbitrary external volume plugins through a well-defined and secure API, providing end users with a broader choice of storage solutions.

Kubernetes has developed its own mechanisms and APIs to implement CSI and support CSI-compliant volume plugins.

The in-tree CSI volume plugin, which is part of Kubelet and is an adapter for interfacing with external CSI drivers, initiates the attachment/detachment processes and invokes the `NodePublishVolume`/`NodeUnpublishVolume` RPCs on the *Node Plugin* to mount/unmount a volume at a specified path.

For creation/deletion and attachment/detachment, external components monitor the Kubernetes API and trigger the `CreateVolume`/`DeleteVolume` and `ControllerPublishVolume`/`ControllerUnpublishVolume` RPCs against a *Controller Plugin*. These components are Kubernetes controllers and perform complex Kubernetes-specific operations. To aid storage providers and alleviate them from any Kubernetes layers, the Kubernetes team has developed and offers these components in the form of sidecar containers meant to be deployed alongside a *Controller Plugin* container in the same Pod.

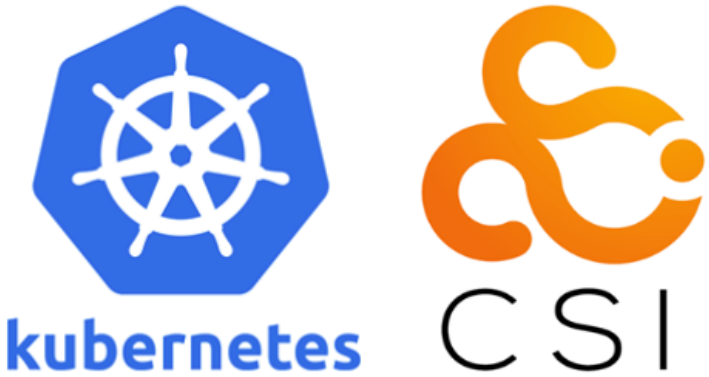


Figure 2.4: *Kubernetes & CSI*

2.4.1 Kubelet to CSI Plugin Communication

Kubelet, through the in-tree CSI volume plugin, communicates with a CSI driver running on the same node via a UNIX domain socket. The CSI driver is required to create a socket under `/var/lib/kubelet/plugins/<plugin-name>/`.

Upon registration, Kubelet invokes the `NodeGetInfo` RPC against the socket. The CSI plugin returns in the `NodeGetInfoResponse` the node identifier (the unique identifier by which the plugin understands and references the node, which might differ from the

Kubernetes Node name) and, if the plugin is topology-aware, the topology information (the set of key-value pairs the plugin uses to topologically identify the node).

Subsequently, Kubelet creates a CSINode object, if one does not exist, and fills it with the information obtained from `NodeGetInfo`. The CSINode resource stores details about all CSI drivers running on the Node. It has the same name as the corresponding Node object and has an owner reference pointing to it. Kubernetes components use the CSINode object to map from the Kubernetes Node name to the driver's node identifier for calls like `ControllerPublishVolume`, and to store and collect the driver's topology information for the `CreateVolume` call in case of topology-aware provisioning.

The in-tree CSI volume plugin

When the AttachDetach controller of `kube-controller-manager` identifies that a Pod utilizing a CSI volume is scheduled on a Node, it triggers the attach operation of the in-tree CSI volume plugin. The in-tree CSI volume plugin then generates a VolumeAttachment object.

The VolumeAttachment resource captures the intention for attaching or detaching a volume on/from a node by a specific CSI plugin. The VolumeAttachment `spec` field includes:

- **`attacher`**: The name of the plugin responsible for the attachment of the volume, as returned by the `GetPluginInfo` RPC.
- **`nodeName`**: The name of the node the volume is to be attached. This is the Kubernetes Node name. The corresponding node identifier as understood by the plugin is obtained from the CSINode object.
- **`source`**: Represents the volume that is to be attached. It is the name of the PersistentVolume.

An external attacher component, deployed alongside a *Controller Plugin* instance, detects the newly created VolumeAttachment object and invokes the `ControllerPublishVolume` RPC of the plugin via a UNIX domain socket.

Upon deleting the same Pod from the Node, the AttachDetach controller triggers the detach operation of the in-tree CSI volume plugin. The in-tree CSI volume plugin deletes

the corresponding VolumeAttachment object. The external attacher component recognizes that the VolumeAttachment object is marked for deletion and invokes the `ControllerUnpublishVolume` RPC of the CSI plugin.

The `SetUp` and `TearDown` methods of the in-tree CSI volume plugin directly invoke the `NodePublishVolume` and `NodeUnpublishVolume` RPCs of the registered CSI plugin. For `NodePublishVolume`, Kubelet creates a unique per-Pod per-volume target path under `/var/lib/kubelet` for the CSI plugin to mount the volume. Following a successful `NodePublishVolume`, Kubelet bind-mounts the volume within the container(s) of the Pod. Upon a successful `NodeUnpublishVolume`, Kubelet deletes the target path.

`node-driver-registrar`

To simplify deployment and registration for a *Node Plugin*, Kubernetes provides the `node-driver-registrar` sidecar container that runs in the same Pod as the plugin and registers the plugin with Kubelet using the kubelet plugin registration mechanism [3].

The `node-driver-registrar` uses two UNIX domain sockets: a registration socket (shared between Kubelet and the `node-driver-registrar`) and a CSI driver socket (shared between all three; CSI driver, Kubelet, and `node-driver-registrar`).

The registration socket is a `hostPath` volume mounted from `/var/lib/kubelet/plugins-registry/` on the host at `/registration/` inside the `node-driver-registrar` container. The `node-driver-registrar` creates a UNIX domain socket under the `/registration` directory and communicates with Kubelet on the host.

The CSI driver socket is a `hostPath` volume mounted from `/var/lib/kubelet/plugins/<driver-name>/` on host inside the CSI driver and `node-driver-registrar` containers. The CSI driver creates a UNIX domain socket under this container mount path. The full host path of the CSI driver socket and the corresponding container path inside the `node-driver-registrar` container are passed as arguments to the `node-driver-registrar`. The `node-driver-registrar` invokes the `GetPluginInfo` RPC on the given CSI endpoint to get the driver's name and informs Kubelet via the registration socket about the CSI driver socket on the host, thus registering the driver with Kubelet.

As the generated target paths of Kubelet for the `NodeStageVolume` and `NodeUnpub-`

[3] https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#device-plugin-registration

lishVolume RPCs are under /var/lib/kubelet/, a containerized *Node Plugin* must also mount this directory within its container. Bidirectional mount propagation is crucial to ensure the plugin receives updates from Kubelet and the plugin's mounts are propagated on the host and detected by Kubelet.

2.4.2 Master to CSI Plugin Communication

The kube-controller-manager of the master node does not directly communicate with a CSI driver through a UNIX domain socket for security reasons. Instead, it communicates with a CSI driver through the Kubernetes API. External controllers monitor the Kubernetes API server for creation, update, and deletion events on specified resource objects and trigger the corresponding RPCs against the CSI endpoint of the driver.

external-provisioner

The provisioning and deleting operations are handled by an external provisioner. The external-provisioner sidecar container runs in the same Pod with a *Controller Plugin* master container, watches the Kubernetes API for PVC objects that refer to the CSI plugin, and invokes the CreateVolume and DeleteVolume RPCs of the plugin via their shared UNIX domain socket. Similar to Kubelet, the external-provisioner learns the name of the plugin through the GetPluginInfo RPC.

More specifically, an administrator creates a StorageClass that refers through the provisioner field to the CSI plugin's name as returned by GetPluginInfo. A user then creates a PVC using this StorageClass. The external-provisioner recognizes that a PVC refers to the CSI plugin through its StorageClass and invokes the CreateVolume RPC of the plugin. If CreateVolume is successful, the external-provisioner creates a PV using the information returned in the CreateVolumeResponse and binds the PV to the PVC.

When the PVC is deleted, the external-provisioner invokes the DeleteVolume RPC and on success, deletes the released PV.

external-attacher

The attaching and detaching operations are handled by an external attacher. The `external-attacher` sidecar container runs in the same Pod with a *Controller Plugin* master container, watches Kubernetes API for VolumeAttachments objects that refer to the CSI plugin, and invokes the `ControllerPublishVolume` and `ControllerUnpublishVolume` RPCs of plugin via their shared UNIX domain socket.

More specifically, the `external-attacher` detects that a VolumeAttachment object, created by the in-tree CSI volume plugin, refers to the external CSI plugin and invokes the `ControllerPublishVolume` RPC of the plugin. If `ControllerPublishVolume` is successful, the `external-attacher` sets the VolumeAttachment's `status.attached` field to true, indicating the successful attachment to the cluster.

When the VolumeAttachment object, safeguarded by a finalizer, is deleted by the in-tree CSI volume plugin, the `external-attacher` detects the `deletionTimestamp` on it and invokes the `ControllerUnpublishVolume` RPC of the plugin. On success, the `external-attacher` removes the finalizer from the VolumeAttachment, enabling its removal from the API server and indicating the successful detachment of the volume.

2.4.3 Kubernetes CSI Sidecar Containers

The complete list of (non-deprecated) Kubernetes CSI sidecar containers is:

- **`external-provisioner`**: Enables the dynamic provisioning and deletion of volumes. Watches for PVC objects and invokes the `CreateVolume` and `DeleteVolume` RPCs against a CSI endpoint.
- **`external-attacher`**: Enables the attachment and detachment of volumes. Watches for VolumeAttachment object and invokes the `ControllerPublishVolume` and `ControllerUnpublishVolume` RPCs against a CSI endpoint.
- **`external-snapshotter`**: Enables the snapshotting of volumes. Watches for VolumeSnapshot CRD objects and invokes the `CreateSnapshot` and `DeleteSnapshot` RPCs operations against a CSI endpoint.
- **`external-resizer`**: Enables expansion of volumes. Watches for edits on PVC objects and invokes the `ControllerExpandVolume` RPC against a CSI endpoint.

- **`node-driver-registrar`**: Registers the CSI driver with Kubelet using the kubelet plugin registration mechanism. Kubelet invokes the Node service RPCs.
- **`livenessprobe`**: Monitors the health of the CSI driver by invoking the `Probe` RPC and reporting it to Kubernetes via the liveness probe mechanism.
- **`external-health-monitor`**: Monitors the health of the volumes by invoking the `ListVolumes` or `ControllerGetVolume` RPCs against a CSI endpoint and reporting Events on PVCs if the health status is abnormal.

2.4.4 Deploying a CSI Plugin on Kubernetes

Kubernetes does not enforce a specific way of packaging and deploying an external CSI plugin. However, it does provide a recommended architecture that facilitates the deployment and operation of a CSI plugin, as illustrated in Figure 2.5.

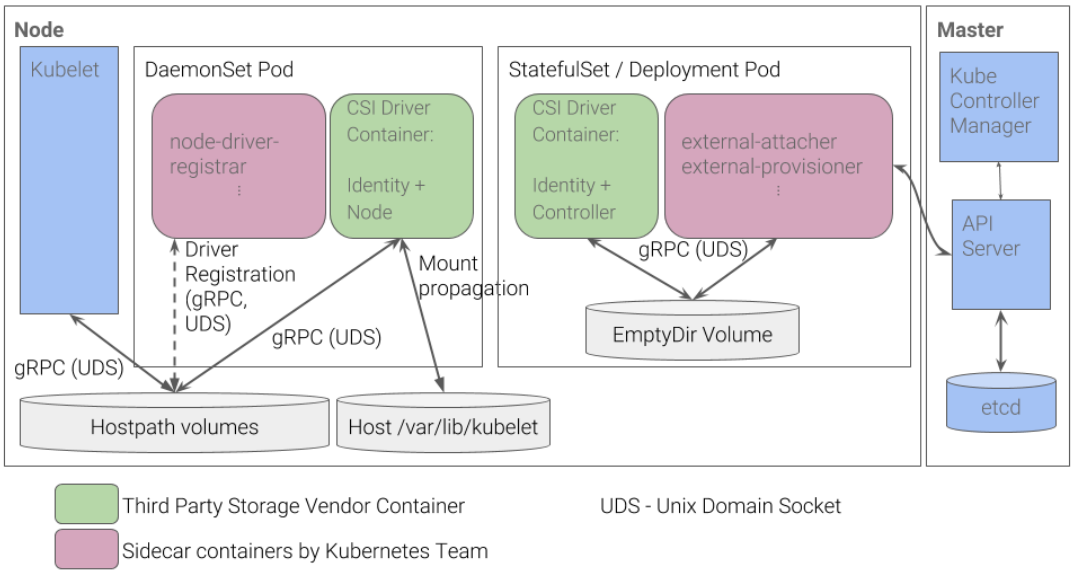


Figure 2.5: *Recommended deployment architecture of CSI plugins on Kubernetes*

[Source: https://github.com/kubernetes/design-proposals-archive/blob/main/storage/container-storage-interface.md#recommended-mechanism-for-deploying-csi-drivers-on-kubernetes]

According to the recommendation, all CSI plugin instances are containerized. One or more *Controller Plugins* are defined in a Deployment or StatefulSet, and all *Node Plugins* result from a DaemonSet. Of course, every plugin must provide the Identity service. For the *Controller Plugin* container, all sidecar containers communicate with the plugin through a UNIX domain socket originating from a shared `emptyDir` volume. For the *Node Plugin* container, registered with Kubelet via the `node-driver-registrar`, at least

two `hostPath` volumes are required for the registration and CSI driver UNIX domain sockets, as described in Section 2.4.1.

2.4.5 Example Walkthrough

In this section, we provide a concise yet end-to-end overview of the CSI implementation in the Kubernetes container orchestrator system, covering the entire volume lifecycle.

2.4.5.1 Creating Volumes

1. A cluster administrator creates a StorageClass object referring to an out-of-tree third-party CSI driver that is installed and running in the Kubernetes cluster. Listing 2.2 shows an example of a StorageClass with `Immediate` volume binding mode, `Delete` reclaim policy, and referencing the `foo.csi.driver` CSI driver.

Listing 2.2: *Example of a StorageClass*

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: example-storageclass
parameters:
  type: fast-ssd
provisioner: foo.csi.driver
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

2. A user creates a PersistentVolumeClaim, in his namespace of choice, claiming a volume of the previously created StorageClass. Listing 2.3 shows an example of a PVC using the StorageClass of the previous example and requesting a 5GiB volume.

Listing 2.3: *Example of a PVC*

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: example-storageclass
  resources:
    requests:
      storage: 5Gi
```

3. The PersistentVolume controller, running as part of the `kube-controller-manager` Pod in the control plane, detects the unbound PVC requiring dynamic provisioning by an out-of-tree CSI volume plugin and sets the `volume.kubernetes.io/storage-provisioner` annotation on the PVC with the driver name from the `provisioner` field of the StorageClass.

4. The external-provisioner sidecar container of the CSI driver detects the unbound PVC with the volume.kubernetes.io/storage-provisioner annotation referring to the CSI driver and (in case of Immediate volume binding mode) invokes the CreateVolume RPC of the driver. The external-provisioner uses parameters from the StorageClass, PVC, and corresponding CSINode objects for the CreateVolumeRequest; the external-provisioner:
   (a) Generates a unique name, pvc-<pvc.uid>, for the CreateVolumeRequest volume identifier.
   (b) Specifies the required capacity of the CreateVolumeRequest based on the requested capacity specified on the PVC.
   (c) Translates the PVC access modes to CSI access modes.
   (d) Specifies mount or block volume, as specified in the PVC.
   (e) Copies the parameters map of the StorageClass in the CreateVolume parameters, along with other optional metadata.
   (f) If the plugin is topology-aware (VOLUME_ACCESSIBILITY_CONSTRAINTS plugin capability), the external-provisioner gathers the topology information of the plugin from the cluster Node and CSINode objects and incorporates it in the CreateVolumeRequest.
5. If CreateVolume returns successfully, the external-provisioner creates a PersistentVolume object to represent the newly provisioned volume and binds it to the PersistentVolumeClaim (pv.claimRef is set to an object reference to the PVC including its name, namespace, and UID). The PV stores the volume_id of the CreateVolumeResponse in spec.csi.volumeHandle.
6. The PersistentVolume controller detects that the newly created PV has a binding reference to the PVC and binds the PVC to the PV (pvc.spec.volumeName is set to pv.name), completing the bi-directional binding process.

2.4.5.2 Deleting Volumes

1. A user deletes the PVC bound to the PV backed by the CSI volume.
2. The external-provisioner of the CSI driver detects the deletion of the PVC and (in case of Delete reclaim policy) invokes the DeleteVolume RPC of the driver using the volume_id of the released PV for the DeleteVolumeRequest.

3. If `DeleteVolume` returns successfully, the `external-provisioner` deletes the released PV object.

2.4.5.3 Attaching Volumes

1. A user creates a Pod that uses the PVC referring to the CSI volume. Listing 2.4 shows an example of a Pod using the PVC of the previous example.

Listing 2.4: *Example of a Pod*

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: app
    image: nginx:stable-alpine
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: vol
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: vol
    persistentVolumeClaim:
      claimName: example-pvc
```

2. The scheduler selects a Node for the Pod.
3. The AttachDetach controller, running as part of the `kube-controller-manager` Pod in the control plane, detects the Pod referring to a CSI volume and scheduled on a Node and triggers the attach operation of the in-tree CSI volume plugin.
4. The in-tree CSI volume plugin creates a new VolumeAttachment object referring to the CSI driver, the CSI volume and scheduled Node.
5. The `external-attacher` sidecar container of the CSI driver detects the VolumeAttachment and invokes the `ControllerPublishVolume` RPC of the driver. The `external-attacher` uses parameters from the VolumeAttachment, PV, and corresponding CSINode objects for the `ControllerPublishVolumeRequest`; the `external-attacher`:
   (a) Translates the Node name of the VolumeAttachment to the `node_id` of the driver found on the corresponding CSINode.
   (b) Dereferences the PV specified on the VolumeAttachment to get the `volume_id` and other metadata returned by the driver in the `CreateVolumeResponse`.
6. If `ControllerPublishVolume` returns successfully, the `external-attacher` updates the VolumeAttachment `status.attached` field to true to indicate the suc-

cessful attachment of the volume.

2.4.5.4 Detaching Volumes

1. A user deletes the Pod that refers to the CSI volume and is scheduled on a Node.
2. The AttachDetach controller detects that the Pod referring to a CSI volume is terminated or deleted and triggers the detach operation of the in-tree CSI volume plugin.
3. The in-tree CSI volume plugin deletes the corresponding VolumeAttachment object which is protected by a finalizer.
4. The `external-attacher` of the CSI driver detects the `deletionTimestamp` on the VolumeAttachment and invokes the `ControllerUnpublishVolume` RPC of the driver using parameters from the VolumeAttachment and corresponding CSINode objects for the `ControllerPublishVolumeRequest`.
5. If `ControllerUnpublishVolume` returns successfully, the `external-attacher` removes the finalizer from the VolumeAttachment to allow for it to be removed from the API and indicate the successful detachment of the volume.

2.4.5.5 Mounting Volumes

1. Kubelet detects that the Pod referring to the CSI volume has been scheduled on the Node and invokes the `WaitForAttach` method of the in-tree CSI volume plugin.
2. The in-tree CSI volume plugin waits on the corresponding VolumeAttachment object until its `status.attached` field is set to true and then returns.
3. Kubelet invokes the `SetUp` method of the in-tree CSI volume plugin which in turn invokes the `NodePublishVolume` RPC of the CSI driver.
4. If `NodePublishVolume` returns successfully, the specified target path where the CSI volume is mounted is further mounted on the Pod container.

2.4.5.6 Unmounting Volumes

1. Kubelet detects that the Pod referring to the CSI volume is terminated or deleted on the Node and invokes the `TearDown` method of the in-tree CSI volume plugin which in turn invokes the `NodeUnpublishVolume` RPC of the CSI driver.

2. If `NodeUnpublishVolume` returns successfully, the specified target path where the CSI volume was mounted is unmounted from the Pod container.

2.5 Union Filesystems

Union filesystems or union mounts, provide a way of overlaying multiple directories and filesystems, known as *branches* or *layers*, to create a new virtual filesystem. This new filesystem appears as a single unified hierarchy that contains the combined contents of its branches. Contents located in the same relative path in two or more branches are viewed together under a single directory in the merged filesystem.

Union mounts can be used to aggregate large storage pools from multiple existing drives for various use cases, including snapshotting, single point of access, and restricting writes to specific segments. A common application of union fileystems is overlaying a small writable filesystem on top of a read-only medium, such as a live CD/DVD. Then, writes to the union mount will be propagated to the writable part of the filesystem, creating the illusion of a fully writable filesystem without altering the base filesystem. This technique is known as copy-on-write (CoW). Union filesystems are also popular in the Docker environment for building Docker images and containers by layering read-only and read/write directories.

Union filesystems were first brought to Linux in 1993 with the introduction of the prototype called Inheriting File System (IFS). Since then, many different implementations have emerged over the years, including UnionFS (2003), aufs (2006) and OverlayFS (2009). The OverlayFS implementation was successfully merged into the mainline Linux 3.18 Kernel in 2014. Similarly, GlusterFS is a solution for union mounting network filesystems.

MergerFS, initially launched in 2014, stands out as an actively maintained open-source FUSE-based implementation, enabling the aggregation of arbitrary branches.

2.5.1 Filesystem in Userspace (FUSE)

Filesystem in Userspace (FUSE) is a framework for enabling non-privileged userspace code to create, export, and manage filesystems outside the kernel.

FUSE simplifies filesystem development considerably, enables rapid prototyping, improves isolation and stability, supports the use of different external libraries, and proceeds at a much higher pace compared to merging filesystem code into the kernel. However, FUSE comes with a significant performance penalty since it has to switch between kernel and user contexts in the critical path.

FUSE consists of two main components:

- **`fuse.ko`**: The FUSE kernel module. This is the bridge to the actual kernel interfaces.
- **`libfuse`**: The userspace shared library. `libfuse` provides functions to mount and unmount FUSE filesystems, read requests from the kernel, and send responses back. It also provides the `fusermount3` utility for mounting by non-root users.

FUSE applications can link to the `libfuse` library and communicate with the FUSE kernel module over the `/dev/fuse` special character device.

To implement a new FUSE filesystem, one has to write a *handler* or *daemon* program that uses `libfuse`. This handler program defines the meaning of the read/write/stat operations for the FUSE filesystem, registers with the kernel, and mounts the filesystem. After that, a user's IO requests on the filesystem will be redirected from the VFS interface to FUSE and then to the handler process. The handler performs the requested operation, returns the response to FUSE, which then forwards it back to VFS and ultimately back to the user.

Figure 2.6 illustrates an example flow-chart diagram of the aforementioned mechanism and user-kernel space switching. A user's list request (`ls -l /tmp/fuse`) is forwarded from VFS to FUSE. FUSE then invokes the handler program (`./hello`) with the passed request (`ls -l /tmp/fuse`). The handler's response is surfaced back to userspace through the kernel, following the reverse path.

FUSE (the kernel-side module) was merged in the mainline Linux 2.6.14 Kernel and its supported platforms include Linux, FreeBSD, OpenBSD, macOs and Windows.

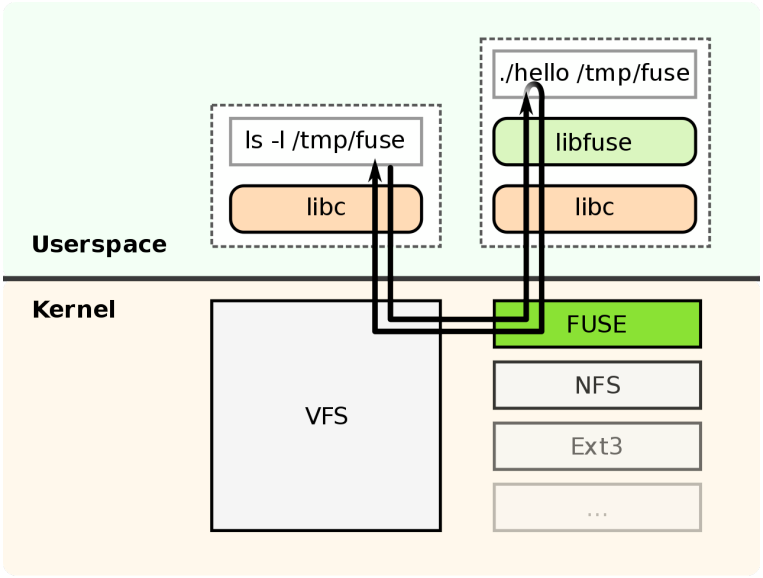


Figure 2.6: *How FUSE works*

[Source: https://en.wikipedia.org/wiki/Filesystem_in_Userspace]

2.5.2 MergerFS

MergerFS is a feature-rich, policy-based union filesystem implementation that utilizes FUSE.

MergerFS operates by logically combining paths, referred to as *branches*, into a new FUSE-based union mount, known as a *pool*, much like a union of sets. Branches can consist of paths from different filesystems, filesystem types, and disk sizes. Alternatively, they can be different paths on the same filesystem. Branches may also already contain data or not.

MergerFS boasts a wide range of features, including:

- Policy-based filesystem access.
- Ability to add or remove filesystems at will.
- Resistance to individual filesystem failure.
- Support for extended attributes (`xattrs`).
- Support for file attributes (`chattr`).
- Runtime configurable (via `xattrs`).
- Works with heterogeneous filesystem types.
- Mixes RO and RW filesystems.
- Hard link copy-on-write.

The command-line syntax of MergerFS is simple:

```
mergerfs -o <options> <branches> <mountpoint>
```

”branches” is a colon (:) delimited list of the paths to merge and ”mountpoint” is the destination path for the resulting union mount.

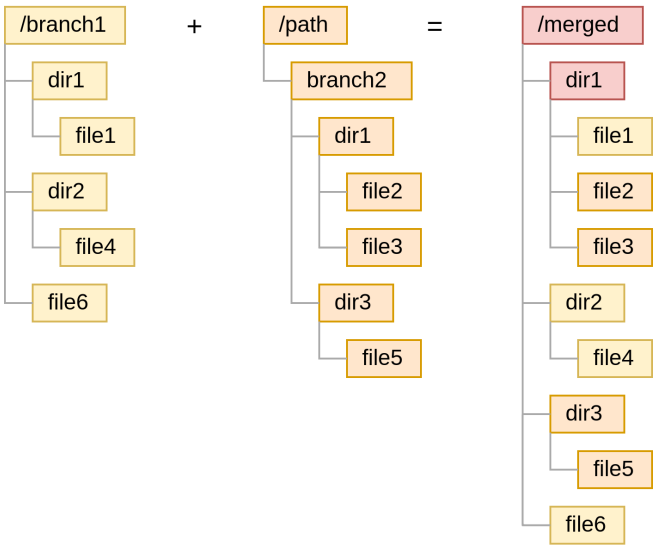
As MergerFS utilizes FUSE, it can also be run by non-root users. Upon successful execution of mergerfs, the MergerFS FUSE filesystem is mounted and the MergerFS FUSE userspace handler operates in the background to service the filesystem. The handler primarily acts as a proxy for its underlying filesystems with additional routing logic for dispatching filesystem functions. MergerFS neither interferes with the data passing through it nor with the underlying filesystems and block devices. Additionally, it does not split or *strip* data across filesystems, as seen in RAID0, but rather divides some behaviors and aggregates others. During normal execution, the FUSE handler process terminates when the MergerFS filesystem is unmounted.

MergerFS groups POSIX filesystem functions (e.g., `creat`, `stat`, `chown`, etc.) into three categories: *action*, *create*, and *search*. Every user IO action on a MergerFS filesystem falls into one of these three categories. Each function and category can be assigned a *policy*, which is the algorithm by which MergerFS selects the underlying destination branch when performing a function. For instance, it may choose the branch with the least or most available space, the first branch where the relative path is found, or a random or semi-random branch.

Policies are also grouped into two categories: *path-preserving* and *non-path-preserving*. Path-preserving policies only consider the branches where the relative path of the function already exists. When a relative path within the pool is created in a branch for the first time (or if it already existed), all subsequent files created under this path will be placed in this branch. For the non-path-preserving counterpart policies, a destination branch is chosen based on the policy and the relative path will be cloned on the branch if it does not already exist. This means that files with the same relative path may be spread across branches. This behavior may not be desirable; for example, a user may wish all their movies to be located under a single `movies` directory when MergerFS is not mounted. In this case, the appropriate path-preserving policy should be selected. The default policy is `epmfs` (existing path, most free space), where, of all the branches where the relative path exists, the one with the most free space is chosen.

The performance of MergerFS incurs an expected overhead due to the FUSE user-kernel space switching and relies on the capabilities and configurations of the devices and filesystems involved in the pool.

```
mergerfs /branch1:/path/branch2 /merged
```




```
/dev/sda on /branch1 type ext4 ...
/dev/sdb on /path/branch2 type xfs ...
branch1:path/branch2 on /merged type fuse.mergerfs ...
```

Figure 2.7: *MergerFS mount example*

3 Design

In this chapter, the provide the design decisions and mechanisms that make up the software application of this thesis.

3.1 Overview

We aim to introduce a Kubernetes volume plugin that exposes union filesystems by aggregating local storage volumes. This plugin integrates with Kubernetes through the Container Storage Interface (CSI) via its CSI driver. We call it Union CSI.

In order to quickly develop a simple working example and realize our concept, we have decided to leverage and integrate existing solutions and constructs. The core idea is that Union CSI, in its current form, functions as a ”meta-plugin”: a plugin that uses other plugins to aggregate the volumes they offer. Union CSI aims to deploy in Kubernetes and integrate seamlessly with an existing storage system (as long as it meets certain requirements we set in this chapter), enabling union mounts of its filesystem instances. This approach not only saves us time and effort, particularly in the context of this thesis, but also eliminates the need to reinvent the wheel by implementing an LVM-based or similar, solution from the ground up.

Union CSI invokes the creation and deletion of volumes from the underlying storage system through the PersistentVolumeClaim Kubernetes API. Kubernetes users should be able to create *extra-large* PVCs (where ”extra-large” implies that the requested storage size greatly exceeds the available disk space of any single node in the cluster). Union CSI

internally creates a set of smaller PVCs, the sum of which satisfies the initial requested capacity. The requested smaller volumes are of the underlying storage system and can fit on capacious local disks across different nodes.

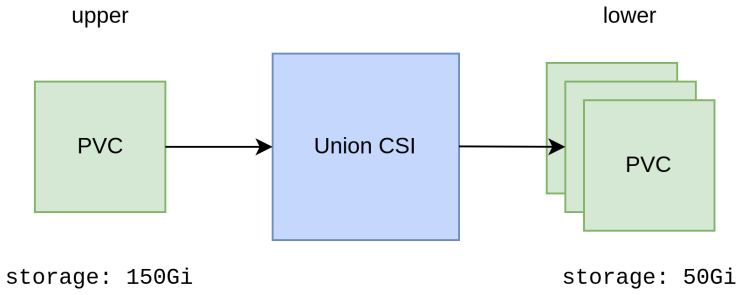
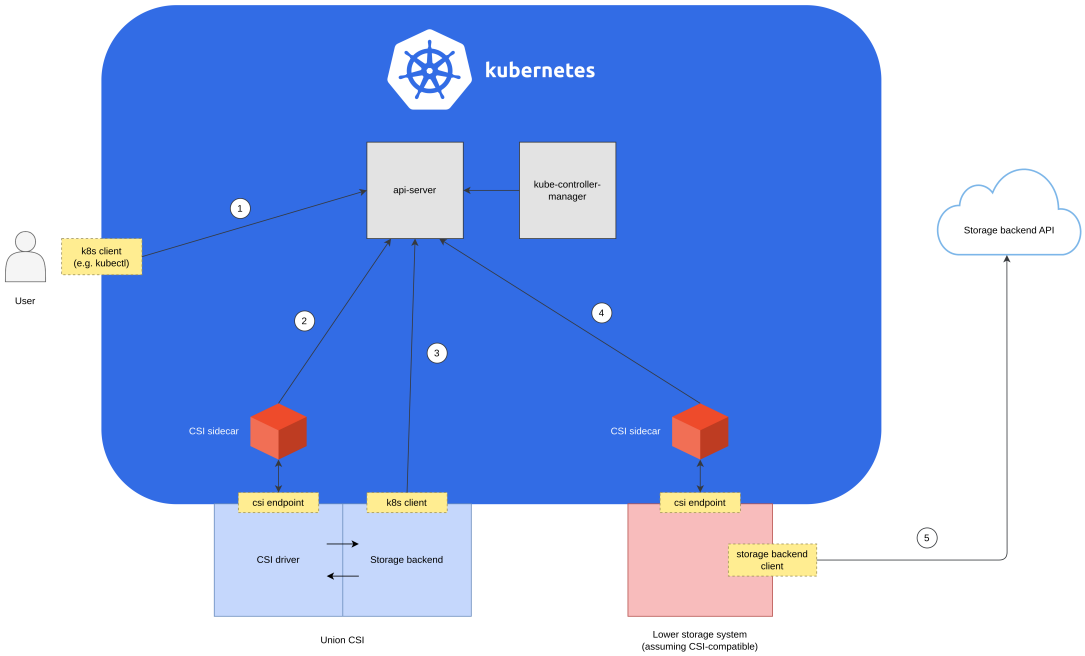


Figure 3.1: *Union CSI manages sets of PVCs*

Another responsibility we aim to delegate from our end to the underlying volume plugin is the ability to remotely access storage. If the volume plugin used by Union CSI leverages local storage from node machines, as opposed to shared storage (think AWS EBS volumes), it should also be capable of linking storage between nodes, creating a storage area network (SAN). This requirement, however, limits our options for storage systems to use to only those that have storage networking capabilities (such as an iSCSI implementation) for accessing remote volumes and transporting data from node to node. While some such plugins do exist, and we deploy one and showcase it in a later chapter, it makes sense for Union CSI to implement this feature, as it is a necessary element to leverage and combine local storage from different nodes. An addition to our future work list for sure.

Union CSI remains agnostic to the underlying storage system's status, operations, or existence. Union CSI is directed towards a specific storage system through the user parameters passed in the `CreateVolume` CSI RPC. This means that the same Union CSI instance can be configured and combined with various existing storage solutions within the same cluster.

Union CSI does not directly communicate with the underlying plugin; communication is mediated through Kubernetes. Union CSI employs the same Kubernetes API that a user would (i.e., PVCs and Pods) to trigger the desired volume operations of the underlying plugin, and observes the results through updates on the API objects.



1 A user makes a request to the Kubernetes API server for a storage operation from the Union CSI plugin and Kubernetes updates the cluster state accordingly. For example, a user creates a PVC using a Union CSI StorageClass and the PersistentVolume controller of kube-controller-manager updates the PVC for an external provisioner.

2 A Kubernetes CSI sidecar container deployed with a Union CSI instance picks up the updated state of Kubernetes and invokes the corresponding RPC of the Union CSI plugin via a gRPC endpoint. For example, the external-provisioner sidecar container detects the updated PVC and invokes the CreateVolume RPC.

3 The Union CSI plugin instance validates the RPC request (CSI driver component) and invokes the Kubernetes API (storage backend component) to retrieve the current state of the requested storage operation and update it if necessary. For example, Union CSI creates a sets of PVCs using a StorageClass of the lower storage system.

4 A Kubernetes CSI sidecar container deployed with a lower storage system instance picks up the updated state of Kubernetes and invokes the corresponding RPC of the lower storage system via a gRPC endpoint. For example, the external-provisioner sidecar container detects one or more PVCs and invokes the CreateVolume RPC. We assume that the storage system integrates with Kubernetes through CSI.

5 The storage system carries out the requested storage operation. For example, the storage system provisions the requested volume(s) on behalf of Union CSI. The storage system (likely) invokes its storage backend to retrieve and update its state, which can be an external API or an extended Kubernetes API.

Figure 3.2: *Union CSI meta-plugin design overview*

Union CSI utilizes MergerFS to pool the volumes provided by the underlying storage system together and provides a FUSE filesystem to workloads. The underlying volumes are either mounted locally to the node where a workload will run or remotely through the network, and in each case, they are managed by their storage system. The routing of files and data written on the FUSE filesystem across the underlying volumes is dictated by the MergerFS policy used. Thus, Union CSI can enable distributed storage but not replicated or striped storage. MergerFS does not shard data across the underlying filesystems.

As Union CSI directly relies on MergerFS for its data path, the same do's and don'ts for MergerFS mounts apply for Union CSI volumes. For some examples of don'ts, please refer directly to the `README.md` file on the MergerFS GitHub repository [1].

3.2 Terminology

Term	Description
CSI Volume Plugin	A third-party CSI-compliant volume plugin that provides storage to containerized workloads in a CO environment through CSI services. *Note*: In Kubernetes, the term "CSI volume plugin" is sometimes used to refer to the in-tree CSI adapter mechanism that Kubernetes uses to interact and integrate with out-of-tree "CSI volume drivers".
CSI Volume Driver	The part of a CSI volume plugin that implements the CSI services and provides them through a gRPC endpoint.
Upper	Adjective for entities, components and resources related to the Union CSI volume plugin, e.g. upper volume plugin, upper volume, upper PVC, upper PV, etc.
Lower	Adjective for entities, components and resources related to the underlying storage system used by the Union CSI plugin, e.g. lower volume plugin, lower volume, lower PVC, lower PV, etc.

[1] https://github.com/trapexit/mergerfs?tab=readme-ov-file#what-should-mergerfs-not-be-used-for

From now on, the entire Union CSI software will be referred to as *Union CSI volume plugin*, *Union CSI plugin*, or simply *Union CSI*. The specific component responsible for implementing and providing the CSI services (CSI driver), that is a subset of the Union CSI plugin, will be referred to as *Union CSI volume driver* or simply *Union CSI driver*. We will also use the terms *upper* and *lower* to help distinguish between the Union CSI and utilized plugins.

3.3 Goals

In terms of the operations and separation of concerns between Union CSI (upper plugin) and underlying storage system (lower plugin), Union CSI **will**:

- Create and delete PersistentVolumeClaims that use a StorageClass of the lower storage system to trigger the provisioning and deletion of its volumes.
- Create and delete Pods that use the PersistentVolumeClaims created by Union CSI for the lower storage system to trigger the attachment/detachment and mounting/unmounting of its volumes.
- Utilize MergerFS within the Pod container to merge the volumes of the lower storage system mounted on the container and mount the union filesystem on the host node as a directory.
- Mount (bind) the host node union filesystem at the target path specified by Kubelet to be further mounted inside consumer containers.

3.4 Non-Goals

In terms of the operations and separation of concerns between Union CSI (upper plugin) and underlying storage system (lower plugin), Union CSI **will not**:

- Directly allocate, manage, or free block storage from the disks attached on the nodes of the cluster. Union CSI relies on the underlying storage system for this functionality.
- Establish, try to detect, or directly make use of a SAN of any kind in the cluster. Union CSI relies on the lower storage system for this functionality. If the lower

storage system is unable to access its volumes from different nodes in the cluster, then Union CSI will fail to schedule a Pod that uses volumes from different nodes.
- Directly communicate with the lower storage system through an API. Union CSI will create, monitor and delete Kubernetes resources (i.e. PVCs, Pods) that invoke the desired volume operations of the lower storage system.

3.5 Design Details

In this section, we delve into the details of the CSI specification, explaining how the Union CSI plugin implements CSI and focusing on the RPCs it supports. We outline how the Union CSI backend uses the Kubernetes API, utilizing both built-in and custom resources, to carry through with creating, persisting, retrieving, deleting, attaching, detaching, mounting, and unmounting lower and upper volumes. Additionally, we cover how a containerized MergerFS can assemble smaller volumes to form and mount a Union CSI volume on a node.

3.5.1 Idempotency

All CSI RPCs must be idempotent. A CSI RPC made with the same parameters must always return the same result. Ensuring this idempotency is the responsibility of the CSI plugin. Some examples are:

- The `CreateVolume` RPC should initially check for the existence of the requested volume. If the volume already exists, the next step is to ensure compatibility with the requested parameters, such as verifying that the requested volume size is not greater than the provisioned volume's size (though it can be lower).
- The `ControllerPublishVolume` RPC should return success if the requested volume is already attached at the given node. Otherwise, it proceeds with the necessary attachment process.
- The `DeleteVolume` RPC should return success if the requested volume does not exist. Whether it was deleted or never existed at all is of no concern.

Union CSI adheres to the idempotency requirements of each implemented RPC that alters the state of the system. See Section 3.6 for examples.

3.5.2 Timeouts

Any of the CSI RPC requests may time out and be retried at a later time. It is solely up to the CO to decide when a call expires, how long to wait to retry the call and the maximum number of retries. Idempotency requirements ensure that a retried call with the same fields continues where it left off when retried or naturally picks up the updates since the last call.

The CSI driver and storage backend API are responsible for the idempotency of the CSI requests. Some storage backends may take a long time to carry out the given volume operation and respond back, leading to the expiration of the related CSI RPC request. The `DEADLINE_EXCEEDED` gRPC error code may be returned by the driver in this case (see Section 3.5.4).

3.5.3 Concurrency

The CO is responsible for ensuring that there is no more than one call ”in-flight” per volume at a given time. However, in some circumstances, the CO might lose state (for example when the CO crashes and restarts), and may issue multiple calls simultaneously for the same volume. The CSI plugin should handle this as gracefully as possible. The `ABORTED` gRPC error code may be returned by the driver in this case (see Section 3.5.4).

Considering this, after consulting existing and well-known CSI drivers and in the context of this first iteration of the Union CSI plugin, any concurrency issues will be left to Kubernetes. Union CSI does not implement a concurrency mechanism.

3.5.4 Errors

All CSI RPCs must return, along with the per-RPC response, a standard gRPC status. Status consists of three fields: `code`, `message` and `details`.

The `code` field must contain a canonical error code used by gRPC. Each RPC defines a set of gRPC error codes that must be returned by the CSI plugin when the specified conditions are met. The CO must recognise and handle all known gRPC error codes and implement the corresponding error recovery behavior for the given code. In ad-

dition to the per-RPC conditions and gRPC error codes, a global set of conditions and corresponding error codes that the CSI plugin must adhere to is shown in Table 3.1.

Condition	gRPC Code	Description	Recovery Behavior
Missing required field	3 `INVALID_ARGUMENT`	Indicates that a required field is missing from the request. More human-readable information MAY be provided in the `status.message` field.	Caller MUST fix the request by adding the missing required field before retrying.
Invalid or unsupported field in the request	3 `INVALID_ARGUMENT`	Indicates that the one or more fields in this field is either not allowed by the Plugin or has an invalid value. More human-readable information MAY be provided in the gRPC `status.message` field.	Caller MUST fix the field before retrying.
Permission denied	7 `PERMISSION_DENIED`	The Plugin is able to derive or otherwise infer an identity from the secrets present within an RPC, but that identity does not have permission to invoke the RPC.	System administrator SHOULD ensure that requisite permissions are granted, after which point the caller MAY retry the attempted RPC.

continued on next page

Table 3.1 – continued from previous page

Condition	gRPC Code	Description	Recovery Behavior
Operation pending for volume	10 `ABORTED`	Indicates that there is already an operation pending for the specified volume. In general the Cluster Orchestrator (CO) is responsible for ensuring that there is no more than one call "in-flight" per volume at a given time. However, in some circumstances, the CO MAY lose state (for example when the CO crashes and restarts), and MAY issue multiple calls simultaneously for the same volume. The Plugin, SHOULD handle this as gracefully as possible, and MAY return this error code to reject secondary calls.	Caller SHOULD ensure that there are no other calls pending for the specified volume, and then retry with exponential back off.
Call not implemented	12 `UNIMPLEMENTED`	The invoked RPC is not implemented by the Plugin or disabled in the Plugin's current mode of operation.	Caller MUST NOT retry. Caller MAY call `GetPluginCapabilities`, `ControllerGetCapabilities`, or `NodeGetCapabilities` to discover Plugin capabilities.

continued on next page

Table 3.1 – continued from previous page

Condition	gRPC Code	Description	Recovery Behavior
Not authenticated	16 `UNAUTHENTICATED`	The invoked RPC does not carry secrets that are valid for authentication.	Caller SHALL either fix the secrets provided in the RPC, or otherwise regalvanize said secrets such that they will pass authentication by the Plugin for the attempted RPC, after which point the caller MAY retry the attempted RPC.

Table 3.1: *CSI general gRPC error scheme*

[Source: https://github.com/container-storage-interface/spec/blob/master/spec.md#error-scheme]

The `message` field must contain a human readable message describing the non-OK errors, and the `details` field must be empty.

Union CSI complies with the above table along with the per-RPC conditions and errors for its error handling behavior for each of its implemented RPCs. See Section 3.6 for examples.

3.5.5 Capabilities

A CSI plugin advertises the RPCs it supports through a capabilities system. Capabilities are enumerations with values that represent different volume features supported by the plugin and that the corresponding RPCs are implemented to access them. There are specific RPCs that return the plugin's capabilities. This system allows the CSI plugin to inform the CO about its functionality and the additional features it supports, aside from basic volume creation, attachment, mounting, and their reverse operations, such as volume snapshotting, cloning, resizing, topology-constrained, or multi-writer volumes.

There is a dedicated RPC for reporting capabilities for each service:

- **Identity service**: `GetPluginCapabilities` RPC
- **Controller service**: `ControllerGetCapabilities` RPC
- **Node service**: `NodeGetCapabilities` RPC

The RPCs of the Identity service are required by all CSI plugins. The capabilities of `GetPluginCapabilities` describe the plugin "as a whole", providing a summary of the capabilities across all its instances. Every instance within the plugin deployment, identified by the name and version returned by the `GetPluginInfo` RPC, must return the exact same set of capabilities. This holds true regardless of the specific per-instance RPCs served. For example, if a plugin instance implements the Controller service, then all instances must report the `CONTROLLER_SERVICE` capability in `GetPluginCapabilitiesResponse`, as well as the supported Controller service capabilities reported in `ControllerGetCapabilitiesResponse`, even instances that are deployed only as a *Node Plugin*.

The capabilities returned by `ControllerGetCapabilities` indicate which optional Controller service RPCs are implemented, while the capabilities returned by `NodeGetCapabilities` indicate which optional Node service RPCs are implemented.

There is no specific capability signaling whether a plugin instance implements the Node service RPCs. In Kubernetes, if a plugin instance is registered with Kubelet through the kubelet plugin registration mechanism, it is assumed to implement the `NodeGetCapabilities` RPC for reporting Node service capabilities, along with the `NodePublishVolume`/`NodeUnpublishVolume` RPCs for mounting/unmounting volumes.

Some examples of capabilities are:

- **`CONTROLLER_SERVICE`** (Identity service): Indicates that the driver implements RPCs of Controller service. The CO can invoke the required Controller service RPCs. To determine which specific optional Controller service RPCs are served, `ControllerGetCapabilities` must be called.
- **`VOLUME_ACCESSIBILITY_CONSTRAINTS`** (Identity service): Indicates that the volumes of the plugin may not be equally accessible from all nodes in the cluster.

If the driver supports this capability, then additional topology information is returned by the `CreateVolume` and `NodeGetInfo` RPCs that the CO must use to ensure that a given volume is accessible from a given node when scheduling workloads.

- **`CREATE_DELETE_VOLUME`** (Controller service): Indicates that the plugin supports dynamic provisioning and deleting volumes. A driver advertising this capability must implement the `CreateVolume` and `DeleteVolume` RPCs.
- **`PUBLISH_UNPUBLISH_VOLUME`** (Controller service): Indicates that the plugin supports attaching and detaching volumes on nodes. A driver advertising this capability must implement the `ControllerPublishVolume` and `ControllerUnpublishVolume` RPCs.
- **`CREATE_DELETE_SNAPSHOT`** (Controller service): Indicates that the plugin is capable of creating volumes from snapshots and deleting them.
- **`STAGE_UNSTAGE_VOLUME`** (Node service): Indicates that the plugin stages its attached volumes at a global mount point on the node before they can be used for subsequent mounts on containers. A driver advertising this capability must implement the `NodeStageVolume` and `NodeUnstageVolume` RPCs.

Union CSI is a *minimal* CSI plugin that focuses on the basic functionalities of creating, attaching, mounting, and volumes. Its supported capabilities per RPC service are outlined in Table 3.2.

RPC service	Supported Capabilities
Identity service	`CONTROLLER_SERVICE`
Controller service	`CREATE_DELETE_VOLUME` `PUBLISH_UNPUBLISH_VOLUME`
Node service	-

Table 3.2: *Union CSI supported capabilities*

3.5.6 Volume Capabilities

CSI supports two types of volumes: *mount* and *block* volumes. Mount volumes appear inside containers as a mounted directory with a specified filesystem, while block volumes appear inside containers as raw block devices.

The type of volume in a CSI RPC request is specified in the `volume_capability` field of the request, particularly in its `access_type` field. If a mount volume is requested, additional fields can be specified, such as the desired filesystem type and mount options. In the case of a block volume request, there are no additional fields.

The `volume_capability` also includes the `access_modes` field, which specifies the access modes of the volume. The list of CSI access modes along their descriptions is shown in Table 3.3.

Access Mode	Description
`SINGLE_NODE_WRITER`	Can only be published once as read/write on a single node, at any given time.
`SINGLE_NODE_READER_ONLY`	Can only be published once as read-only on a single node, at any given time.
`MULTI_NODE_READER_ONLY`	Can be published as read-only at multiple nodes simultaneously.
`MULTI_NODE_SINGLE_WRITER`	Can be published at multiple nodes simultaneously. Only one of the nodes can be used as read/write. The rest will be read-only.
`MULTI_NODE_MULTI_WRITER`	Can be published as read/write at multiple nodes simultaneously.
`SINGLE_NODE_SINGLE_WRITER`	Can only be published once as read/write at a single workload on a single node, at any given time. SHOULD be used instead of `SINGLE_NODE_WRITER` for COs using the experimental `SINGLE_NODE_MULTI_WRITER` capability.
`SINGLE_NODE_MULTI_WRITER`	Can be published as read/write at multiple workloads on a single node simultaneously. SHOULD be used instead of `SINGLE_NODE_WRITER` for COs using the experimental `SINGLE_NODE_MULTI_WRITER` capability.

Table 3.3: *CSI volume access modes*

Unlike driver capabilities, volume capabilities are not programmatically advertised by the CSI driver, as explained in Section 3.5.5. Instead, COs will directly pass the requested volume types and access modes to CSI drivers. If a volume is requested with unsupported capabilities or if the requested capabilities are incompatible with an already provisioned volume, the plugin must reject the request, returning the appropriate gRPC error code. End users should refer to their storage provider's documentation to understand and choose from the volume capabilities offered by the plugin.

Some RPCs, such as `CreateVolume` and `ValidateVolumeCapabilities`, include an array of possible volume capabilities in the `volume_capabilities` field. The CSI driver must validate each of them and ensure that all are supported. The following RPCs specify either `volume_capability` or `volume_capabilities` and require validation:

- `CreateVolume`
- `ControllerPublishVolume`
- `ValidateVolumeCapabilities`
- `GetCapacity`
- `NodeStageVolume`
- `NodePublishVolume`

The Union CSI plugin exclusively supports union mounts of volumes, and as a result, it can only accommodate mount volumes. Requests for block volumes are rejected, returning the `INVALID_ARGUMENT` gRPC error code along with a related error message in `status.message`. In this version of the Union CSI plugin, the lower volumes are also exclusively mount volumes, requested with the `spec.volumeMode` set to `Filesystem` for the lower PersistentVolumeClaims (or omitted, as `Filesystem` is the default).

Additionally, the `SINGLE_NODE_WRITER` access mode is the most basic and straightforward to implement. Union CSI only supports `SINGLE_NODE_WRITER` and rejects requests with invalid access modes, again returning the `INVALID_ARGUMENT` gRPC error code along with a related error message.

Considering this, Union CSI creates lower PersistentVolumeClaim objects using the `ReadWriteOnce` volume access mode. This means that the lower plugin must support the `SINGLE_NODE_WRITER` or the niche `SINGLE_NODE_MULTI_WRITER` volume capabilities. In practice, all CSI drivers support the `SINGLE_NODE_WRITER` volume capability,

offering single node, one-to-one, read/write volumes. However, if the lower volume plugin does not provision volumes for the Kubernetes `ReadWriteOnce` access mode, the PVCs created by Union CSI will remain unbound and the Union CSI volume will not be created.

The mapping of Kubernetes to CSI access modes can be found in the `kubernetes-csi/csi-lib-utils` GitHub repository [2].

3.5.7 Topology

Some storage systems may choose to limit or be incapable of providing volumes uniformly across all nodes in a cluster. Such storage systems may constrain their volumes to specific subsets of nodes based on topological considerations, such as ”region” and ”zone”.

To support topology-aware volume plugins in expressing, reporting, and accepting topology information for seamless integration with COs, the CSI specification facilitates the following:

- **Topology-awareness capability reporting**: The CSI driver can signal that the volume plugin is topology-aware by advertising the `VOLUME_ACCESSIBILITY_CONSTRAINTS` capability in the `GetPluginCapabilitiesResponse`.
- **Node topology information reporting**: The CSI driver can report the topology of a node through opaque key-value pairs in the `NodeGetInfoResponse`. For example, node N exists in ”region”=”R1” and ”zone”=”Z2”.
- **Volume topology information reporting**: The CSI driver can report the topology of a volume in the `CreateVolumeResponse`. For example, volume V is accessible in ”region”=”R1” and ”zone”=”Z1”, or ”region”=”R1” and ”zone”=”Z2”.
- **User specification of desired topology**: CO users can influence or specify the desired topology of a new volume in CreateVolumeRequest. For example, they may wish to provision volume V in ”region”=”R1” and ”zone”=”Z1”, or ”region”=”R1” and ”zone”=”Z2”.

In Kubernetes, with the assistance of the `external-provisioner` sidecar, topology-aware provisioning is implemented as follows:

[2] https://github.com/kubernetes-csi/csi-lib-utils/blob/v0.16.0/accessmodes/access_modes.go

1. The CSI plugin registers with Kubelet through the kubelet plugin registration mechanism.
2. The CSI plugin communicates the topological constraints of a node in `NodeGetInfoResponse.accessible_topology` of the `NodeGetInfo` RPC.
3. Kubelet creates a new CSINode object (if one does not already exist) referencing the Node object of the CSI plugin instance, populates it with the information returned by `NodeGetInfo` and stores the topology key-value pairs on the Node as labels. The `external-provisioner` later retrieves this topology information from the CSINode and Node objects.
4. When a user triggers dynamic provisioning by creating a PersistentVolumeClaim, `external-provisioner` prepares the topology information in `CreateVolumeRequest.accessibility_requirements` of the `CreateVolume` RPC. The `external-provisioner` collects all the topology information of the cluster reported by the CSI plugin by aggregating the CSINode and Node objects. The arrangement logic of `accessibility_requirements` is influenced by factors such as the volume binding mode of the StorageClass (`Immediate` or `WaitForFirstConsumer`), the `allowedTopologies` field of the StorageClass, and command-line arguments of `external-provisioner`. Exactly how the preparation of `accessibility_requirements` is controlled is documented in the source code repository of the `external-provisioner` [3].
5. In the `CreateVolume` RPC, the CSI plugin must honor the topology constraints specified in `CreateVolumeRequest.accessibility_requirements`. The actual topological segment(s) of the provisioned volume returned in `CreateVolumeResponse.volume.accessible_topology` must be a subset of `CreateVolumeRequest.accessibility_requirements`.
6. If the `CreateVolume` RPC succeeds and topology information is returned, the `external-provisioner` stores this information in the PersistentVolume object in the `spec.nodeAffinity` field. This ensures that Pods using this volume will only get scheduled on Nodes matching the node affinity of the PV.

To implement topology in Union CSI, both the upper and lower plugins must recognize and advertise the same set of topology key-value pairs. This requires Union CSI to have a mechanism for detecting topology information reported by the lower plugin and

[3] https://github.com/kubernetes-csi/external-provisioner#topology-support

returning the same information per node in the `NodeGetInfoResponse`. However, as outlined in Section 3.1, Union CSI is designed to integrate with various lower storage systems that support the required features, and it is not aware of the specific lower storage system in use at any given time. Additionally, Union CSI lacks a means to propagate topologies passed in `CreateVolumeRequest` to the lower plugin through the created PersistentVolumeClaims, as the PVC API lacks relevant fields to restrict provisioning to specific topologies.

For these reasons, Union CSI, in its current form, is not a topology-aware volume plugin and does not support the `VOLUME_ACCESSIBILITY_CONSTRAINTS` capability. The lower volume plugins paired with Union CSI must also be topology-agnostic, ensuring their volumes are accessible on all nodes of the cluster. If the lower volume plugin used is topology-aware, Union CSI may exhibit unexpected behavior, potentially causing Pods using its volumes to become unschedulable.

3.5.8 RPC Interface

The Union CSI driver implements the following CSI RPCs:

RPC service	Supported RPCs

Identity service	`GetPluginInfo`
	`Probe`
	`GetPluginCapabilities`

Controller service	`CreateVolume`
	`DeleteVolume`
	`ControllerPublishVolume`
	`ControllerUnpublishVolume`
	`ControllerGetCapabilities`

Node service	`NodePublishVolume`
	`NodeUnpublishVolume`
	`NodeGetInfo`
	`NodeGetCapabilities`

Table 3.4: *Union CSI supported RPCs*

Union CSI does not support and implement the `NodeStageVolume`/`NodeUnstageVolume` RPCs. A Union CSI volume (a MergerFS mount) is attached and staged (mounted) on a Node during the `ControllerPublishVolume` RPC. The volume is staged under the `/var/lib/union-csi` storage area of Union CSI on the host to be bind-mounted within a Pod container during `NodePublishVolume`.

`GetPluginInfo`: Returns the name of Union CSI driver, `union.csi.union.io`, and its version, `dev`, in the `GetPluginInfoResponse`. A StorageClass referring to the Union CSI plugin will have this name in the `provisioner` field.

`Probe`: Returns an empty `ProbeResponse`, signifying that the driver instance is ready.

`GetPluginCapabilities`: Returns the `CONTROLLER_SERVICE` plugin capability.

`CreateVolume`: Creates the lower PersistentVolumeClaims that utilize the underlying volume plugin. Returns the passed volume identifier of `CreateVolumeRequest.name` in `CreateVolumeResponse.volume.volumd_id`.

`DeleteVolume`: Deletes the lower PersistentVolumeClaims created in `CreateVolume`. Returns an empty `DeleteVolumeResponse`.

`ControllerPublishVolume`: Creates a Pod that uses the lower PersistentVolumeClaims and assigns it to the specified Node. The MergerFS container of the Pod merges the lower volumes and mounts the MergerFS filesystem on a `hostPath` volume mounted with bidirectional mount propagation. The filesystem is propagated on the host at `/var/lib/union-csi/volumes/<volume_id>/merged/`. Returns an empty `ControllerPublishVolumeResponse`.

`ControllerUnpublishVolume`: Deletes the Pod that uses the lower PersistentVolumeClaims. The MergerFS container unmounts the filesystem before exiting. Returns an empty `ControllerUnpublishVolumeResponse`.

`ControllerGetCapabilities`: Returns the `CREATE_DELETE_VOLUME` and `PUBLISH_UNPUBLISH_VOLUME` controller capabilities.

`NodePublishVolume`: Bind-mounts the MergerFS filesystem at `/var/lib/union-csi/volumes/<volume_id>/merged` on the specified target path. Returns an empty `NodePublishVolumeResponse`.

`NodeUnpublishVolume`: Unmounts the MergerFS filesystem from the specified target

path. Returns an empty `NodeUnpublishVolumeResponse`.

`NodeGetInfo`: Returns the node identifier in `NetGetInfoResponse.node_id`. It is the name of the Node object on which the driver instance is running. A Union CSI plugin container running on a node obtains the Node name through the `NODE_NAME` environment variable that is set in the Pod manifest to `spec.nodeName`.

`NodeGetCapabilities`: Returns an empty `NodeGetCapabilitiesResponse` since there are no supported node capabilities to report.

Henceforth, we will use the term *"Union CSI Controller"* to refer to a Union CSI plugin instance that implements the Controller service, and *"Union CSI Node"* for an instance that implements the Node service.

3.5.9 Capacity Splitting & Volume Scheduling

Ideally, Union CSI would have a mechanism to monitor disk space usage on each node where a *Union CSI Node* is deployed. This mechanism would enable Union CSI to track the total, used, and available storage space and make informed decisions about appropriate size splittings. However, designing and implementing such a mechanism would be complex. Additionally, it would require Union CSI to synchronize with and follow the node disk pools managed by the lower storage system to ensure that the two plugins share a common view of disk storage usage. Otherwise, any efforts for disk space tracking and smart splits made by Union CSI would be misaligned and ultimately in vain. This path is not a viable option.

As a result, we have chosen a more straightforward approach. Regardless of the requested size, Union CSI proceeds with a "blind" halving. For example, a user's PVC claiming a 100GiB volume with Union CSI always results in two lower 50GiB PVCs made by the plugin, irrespective of storage availability. It is up to the lower plugin to check for availability and schedule the lower volumes if possible. A more flexible configuration would be provided by a ConfigMap volume on the *Union CSI Controller* Pod that specifies the number of replica PVCs to create and thus the current split quantum. The *Union CSI Controller* could watch the ConfigMap for updates, enabling the user to define more fine-grained or larger splits at runtime in case some of the disks are running out of space or new disks or nodes are added to the cluster.

However, even with configurable splittings, Union CSI does not have the means to control or influence the scheduling of the lower volumes. Even if Union CSI could successfully track storage usage, Union CSI simply creates smaller PVCs for the lower plugin to handle, and the PersistentVolumeClaim API does not specify target disks or nodes.

Additionally, consider the scenario where the lower plugin schedules and allocates the two lower volumes on the same node disk that has enough available space for both. Union CSI cannot predict nor prevent this scenario. In this case, a single volume of the original capacity was probably possible by the lower plugin in the first place, and nothing was achieved by incorporating Union CSI.

At this stage, the only way for a user to indirectly affect scheduling and yield a practical use case out of Union CSI is to intentionally request a big enough volume that cannot be placed on a single disk in the cluster, but two volumes half the size can be placed on different disks (and possibly different nodes), and these volumes can be accessed from anywhere in the cluster.

While this design state of Union CSI is static and inflexible, it allows us to make progress toward our goals without hindering the potential for a powerful use case. Consequently, its results will help us validate our assumptions and iterate with a more sophisticated approach in future versions.

3.5.10 Namespace

Every component and resource that is part of the Union CSI Kubernetes deployment, or is directly created by it at the plugin's runtime and is not cluster-scoped, is to be created in the plugin's dedicated namespace. The namespace of the Union CSI plugin is named `union`.

3.5.11 StorageClass

The Union CSI StorageClass currently supports the following parameters:

Parameter	Values	Default	Description
`lowerStorageClassName`			The StorageClass name of the lower volume plugin. This value is used in the `spec.storageClassName` field of the lower PVCs created by Union CSI. If the parameter is missing then the `spec.storageClassName` is omitted by the plugin and the lower PVCs use the default StorageClass of the cluster.

Table 3.5: *Union CSI StorageClass parameters*

The parameters of the StorageClass are passed in the `CreateVolumeRequest.parameters` map. Presently, the sole parameter of the StorageClass is `lowerStorageClassName`, specifying the name of the lower StorageClass to be used for PersistentVolumeClaim objects generated by Union CSI. Prior to users creating PVCs of this StorageClass, cluster administrators must ensure that a lower StorageClass with an identical name is configured and a corresponding lower volume plugin is installed and operational within the cluster.

The StorageClass of the Union CSI plugin uses the `Immediate` volume binding mode so dynamic provisioning commences as soon as a PersistentVolumeClaim is created by the user. `Immediate` mode is preferred because Union CSI currently lacks topology awareness, so there is no reason to defer the provisioning until a Pod is scheduled on a Node. It is important to note that the actual provisioning is handled by the lower volume plugin, and its StorageClass can be configured with either `Immediate` or `WaitForFirstConsumer` mode.

The reclaim policy of the StorageClass is set to `Delete`, enabling the removal of an upper PersistentVolume when a user deletes the upper PVC bound to the PV.

Listing 3.1 shows an example of a Union CSI StorageClass specifying a lower StorageClass called `lower-storageclass`.

Listing 3.1: *Example of a Union CSI StorageClass*

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: union-storage
provisioner: union.csi.union.io
parameters:
  lowerStorageClassName: lower-storageclass
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Figure 3.3 illustrates the relationship between the Union CSI plugin and the lower storage system through their respective StorageClass objects.

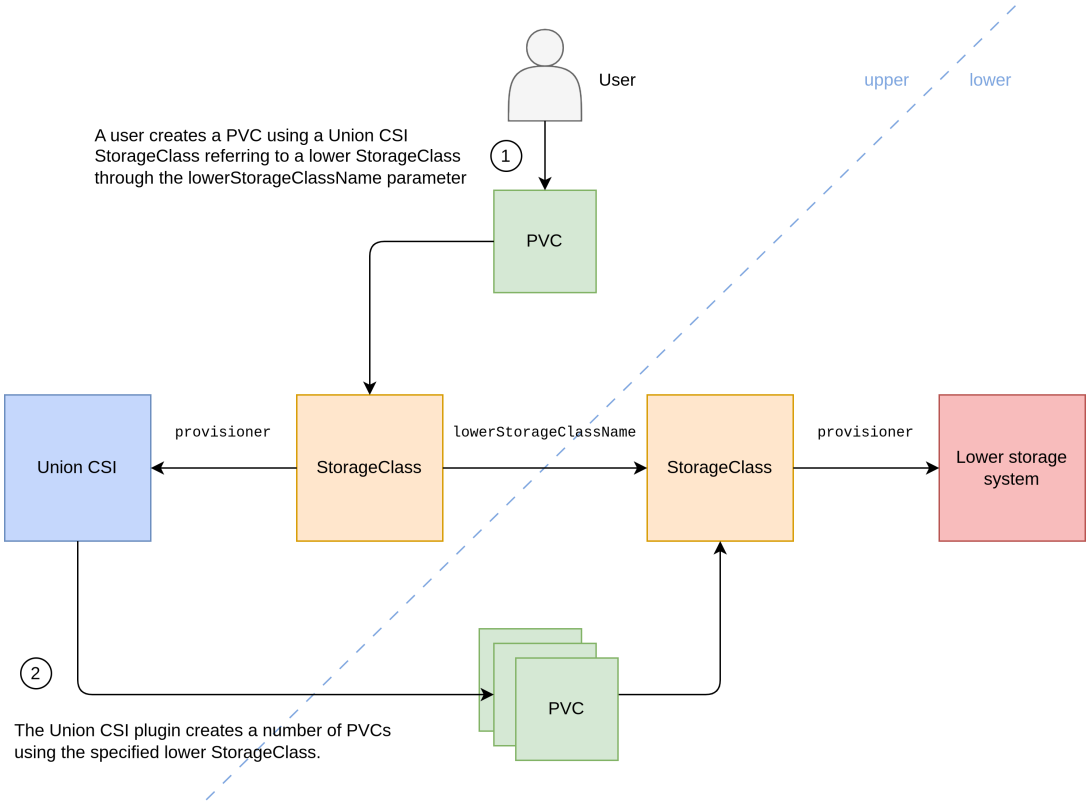


Figure 3.3: *The Union CSI StorageClass and its relationship with a lower storage system*

3.5.12 Sense of Volume

CSI drivers communicate with a storage backend to propagate the requests made by the CO related to volume operations and queries. This storage backend is implemented by the storage provider and exposes an API to access the offered storage solution and services. Additionally, the storage backend is responsible for keeping track of the provisioned volumes, their characteristics, lifecycle state, health, and other relevant attributes.

COs suggest, through CSI, a unique identifier to reference a soon-to-be provisioned volume in the `name` field of `CreateVolumeRequest` of the `CreateVolume` RPC. CSI plu-

gins can choose to use this identifier as is or generate a new one that is more suitable for triggering their storage API and storing and using it to identify the volume.

Union CSI, through MergerFS, provides FUSE-based filesystem volumes that combine filesystem instances of the lower storage system. Union CSI needs a way, given a unique identifier passed in CSI RPCs like `CreateVolume` and `ControllerPublishVolume`, to easily discover if there is a Union CSI volume created in the cluster, quickly access additional information regarding it, and maintain state. Union CSI needs its storage backend.

Given the admission of Kubernetes as the targeted CO of Union CSI, its extensible API, and convenient etcd key-value store, opting for the *Operator Pattern* to extend Kubernetes with CustomResourceDefinitions (CRDs) and controllers was an easy decision.

Many Kubernetes native applications solve problems using *operators*. Developers can host their declarative API on Kubernetes, describe their use cases and behaviors through custom resources and CRDs, and design controllers to implement this behavior and manage the custom resource instances. In addition, the rich ecosystem of Kubernetes libraries and tools enables rapid building and publishing of the API.

Thus, we designed and developed a custom resource called VolumeSplit. The use of the VolumeSplit API solved many of the difficulties developing this CSI volume plugin, including:

- **Single source of truth**: The existence of a VolumeSplit instance implies the existence of a Union CSI volume.
- **Storing and retrieving data**: The VolumeSplit spec consists of fields similar to those of the `CreateVolumeRequest`. When a `CreateVolume` call is made on the Union CSI plugin, a VolumeSplit object is populated with values from the `CreateVolumeRequest` and created on the Kubernetes API. Subsequent `CreateVolume` calls for the same volume result in retrieving the corresponding VolumeSplit object from the etcd. Then, its parameters are compared with those in the `CreateVolumeRequest` to ensure idempotency.
- **Sets of PersistentVolumeClaims**: Union CSI creates smaller PVCs in response to a user's PVC. A VolumeSplit object defines the desired number of PVCs with

identical specifications. A custom controller manages the VolumeSplits and their owned PVC replicas for the Union CSI plugin in the background.

VolumeSplit API

The new VolumeSplit custom resource defined in Go is shown in Figure 3.2.

Listing 3.2: *The VolumeSplit API resource*

```
// VolumeSplit is the Schema for the volumesplits API
type VolumeSplit struct {
    metav1.TypeMeta
    metav1.ObjectMeta

    Spec   VolumeSplitSpec
    Status VolumeSplitStatus
}

// VolumeSplitSpec defines the desired state of VolumeSplit
type VolumeSplitSpec struct {
    // Capacity is the desired minimum amount of storage size.
    // +optional
    Capacity *resource.Quantity

    // Number of desired PersistentVolumeClaims.
    // This is a pointer to distinguish between explicit zero and not specified.
    // Defaults to 1.
    // +optional
    Replicas *int32

    // Template describes the PersistentVolumeClaims that will be created.
    Template corev1.PersistentVolumeClaimTemplate
}

// VolumeSplitStatus defines the observed state of VolumeSplit
type VolumeSplitStatus struct {
    // AccessModes contains the actual access modes of the underlying volumes
    // backing the PersistentVolumeClaims managed by this VolumeSplit.
    // It is the intersection of status.accessModes for all the PersistentVolumeClaims.
    AccessModes []corev1.PersistentVolumeAccessMode

    // Capacity is the actual total capacity of the underlying volumes
    // backing the PersistentVolumeClaims managed by this VolumeSplit.
    // It is the sum of status.capacity.storage for all the PersistentVolumeClaims.
    // If nil it means that every status.capacity is nil.
    // +optional
    Capacity *resource.Quantity

    // Total number of PersistentVolumeClaims managed by this VolumeSplit.
    Replicas int32

    // Total number of PersistentVolumeClaims managed by this VolumeSplit that have ”Bound” ↩
        phase.
    BoundReplicas int32

    // Conditions contain the latest available observations of a VolumeSplit’s current state.
    Conditions []VolumeSplitCondition
}

// VolumeSplitCondition describes the state of a VolumeSplit at a certain point.
type VolumeSplitCondition struct {
    // Type of VolumeSplit condition
    Type VolumeSplitConditionType
    // Status of the condition, one of True, False, Unknown.
    Status corev1.ConditionStatus
    // The last time the condition transitioned from one status to another.
    // +optional
    LastTransitionTime metav1.Time
    // The reason for the condition’s last transition.
    // +optional
    Reason string
    // A human readable message indicating details about the transition.
    // +optional
    Message string
}

// VolumeSplitConditionType is a condition of a VolumeSplit.
type VolumeSplitConditionType string

const (
```

```
    // Ready means that:
    //
    // The actual number of replicas is equal to the desired number.
    // All replicas have "Bound" phase, i.e. each PVC is bound to a PV.
    // If spec.capacity is non-nil, status.capacity is greater than or equal to spec.capacity.
    // status.accessModes array is non-empty.
    VolumeSplitReady VolumeSplitConditionType = "Ready"
    // Pending means that the actual number of replicas is equal to the desired number
    // and at least one replica has "Pending" phase, i.e. at least one PVC is not bound to a PV.
    VolumeSplitPending VolumeSplitConditionType = "Pending"
    // Progressing means that the actual number of replicas has yet to meet the desired number.
    VolumeSplitReplicaProgressing VolumeSplitConditionType = "ReplicaProgressing"
    // ReplicaFailure means that a replica failed to be created or deleted.
    VolumeSplitReplicaFailure VolumeSplitConditionType = "ReplicaFailure"
)
```

The VolumeSplit is a namespaced resource defined under the `union.io` group and the `v1alpha1` version to indicate its experimental state. This means that the Kubernetes API endpoint for VolumeSplit objects is `/apis/union.io/v1alpha1/namespaces/<namespace>/volumesplits.union.io/<name>`.

The VolumeSplit API specifies replicas of PVCs in a manner similar to how a ReplicaSet specifies replicas of Pods. The `replicas` field specifies the desired number of PVCs and the `template` field provides the `metadata` and `spec` for the created PVCs. It is worth noting that, in its current form, VolumeSplit does not incorporate a label selector and does not acquire existing PVCs through one.

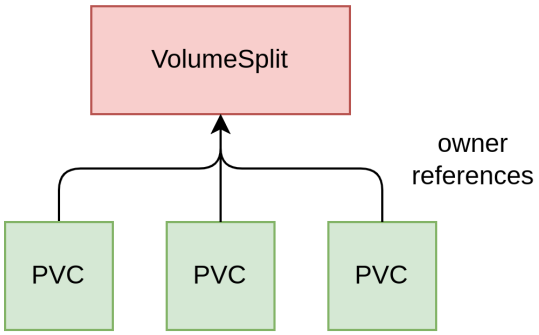


Figure 3.4: *The VolumeSplit custom resource and its replica PVCs*

Figure 3.3 illustrates an example of a VolumeSplit that specifies three replica PVCs of 50GiB each belonging to a StorageClass called `lower-storageclass`.

Listing 3.3: *Example of a VolumeSplit*

```
apiVersion: union.io/v1alpha1
kind: VolumeSplit
metadata:
  name: example-volumesplit
  namespace: union
spec:
  capacity: 150Gi
```

```
  replicas: 3
  template:
    spec:
      accessModes:
      - ReadWriteOnce
      storageClassName: lower-storageclass
      resources:
        requests:
          storage: 50Gi
```

The VolumeSplit `status` uses a `conditions` array following the standard conventions for Kubernetes API `status` properties. Right now, the following condition types exist:

- **`Ready`**:
  - All desired PVCs have been created.
  - All PVCs are bound to a PV.
  - Total actual capacity summed over bound PVCs satisfies the specified capacity (`status.capacity` is greater than or equal to `spec.capacity`).
  - All PVCs have at least one common actual access mode (`status.accessModes` array is non-empty).
- **`Pending`**: All desired PVCs have been created and at least one of them is not bound to a PV yet.
- **`ReplicaProgressing`**: The number of existing PVCs has not reached the desired number.
- **`ReplicaFailure`**: A PVC failed to get created or deleted.

The `Pending` condition is useful to indicate that all the desired PVCs have been created by the VolumeSplit controller but their StorageClass has `WaitForFirstConsumer` volume binding mode. It is like an any-Pending PVC phase, aggregated by an OR on `status.phase` for the `Pending` value of all the PVCs. This condition (coupled with the right `reason` value) can signal to clients waiting for the provisioning of all the volumes claimed by the PVCs that the provisioning process cannot further progress at this point until the PVCs are used by a Pod.

The Union CSI plugin focuses on the `Pending` and `Ready` conditions of the VolumeSplit to confirm that the ”volume” is ready for the attachment process. Additional condition types might be added in the future to include attachment status reports (i.e., Pods using the PVCs).

VolumeSplit objects are meant to be created by the *Union CSI Controller* in the `union`

namespace during the `CreateVolume` RPC. How a VolumeSplit instance is constructed from a `CreateVolumeRequest` is shown in Algorithm 8.

Algorithm 8: How a VolumeSplit is made from a `CreateVolumeRequest`

`name` ← "`split-<CreateVolumeRequest.name>`"
`spec.replicas` ← 2 (see Section 3.5.9)
`spec.capacity` ←
 `CreateVolumeRequest.capacity_range.required_bytes`
`spec.template.spec.accessModes` ← `[ReadWriteOnce]` (see Section 3.5.6)
`spec.template.spec.resources.requests.storage` ←
 `CreateVolumeRequest.capacity_range.required_bytes`/2 (see Section 3.5.9)
if "`lowerStargeClassName`" *key exists in*
 `CreateVolumeRequest.parameters` **then**
  `spec.template.spec.storageClassName` ←
   `CreateVolumeRequest.parameters["lowerStorageClassName"]`
else
  `spec.template.spec.storageClassName` ← `nil` (the replica PVCs will use the default StorageClass of the cluster)

The VolumeSplit controller manages VolumeSplit objects and their replica PVCs. The controller creates the required number of PVCs in the same namespace as the VolumeSplit object. Since Union CSI always creates VolumeSplit objects in the `union` namespace, the lower PVC objects will also live in `union`. Notice how the user PVC (upper) can be in any namespace specified by the user, but the lower PVCs created by Union CSI to shard the initial claim are in the plugin's namespace. How this cross-namespace storage finally becomes available to the end user is made clear in Section 3.5.13.

VolumeSplit controller

The VolumeSplit controller is responsible for creating the desired number of PVCs specified in the `replicas` field of a VolumeSplit object using the PVC template in `spec.template`. The PVCs are created with the `metadata.ownerReferences` field set to the owner VolumeSplit object and with `blockOwnerDeletion` set to true, so the owner VolumeSplit deletion is blocked until the owned PVCs are deleted in case of foreground cascading deletion.

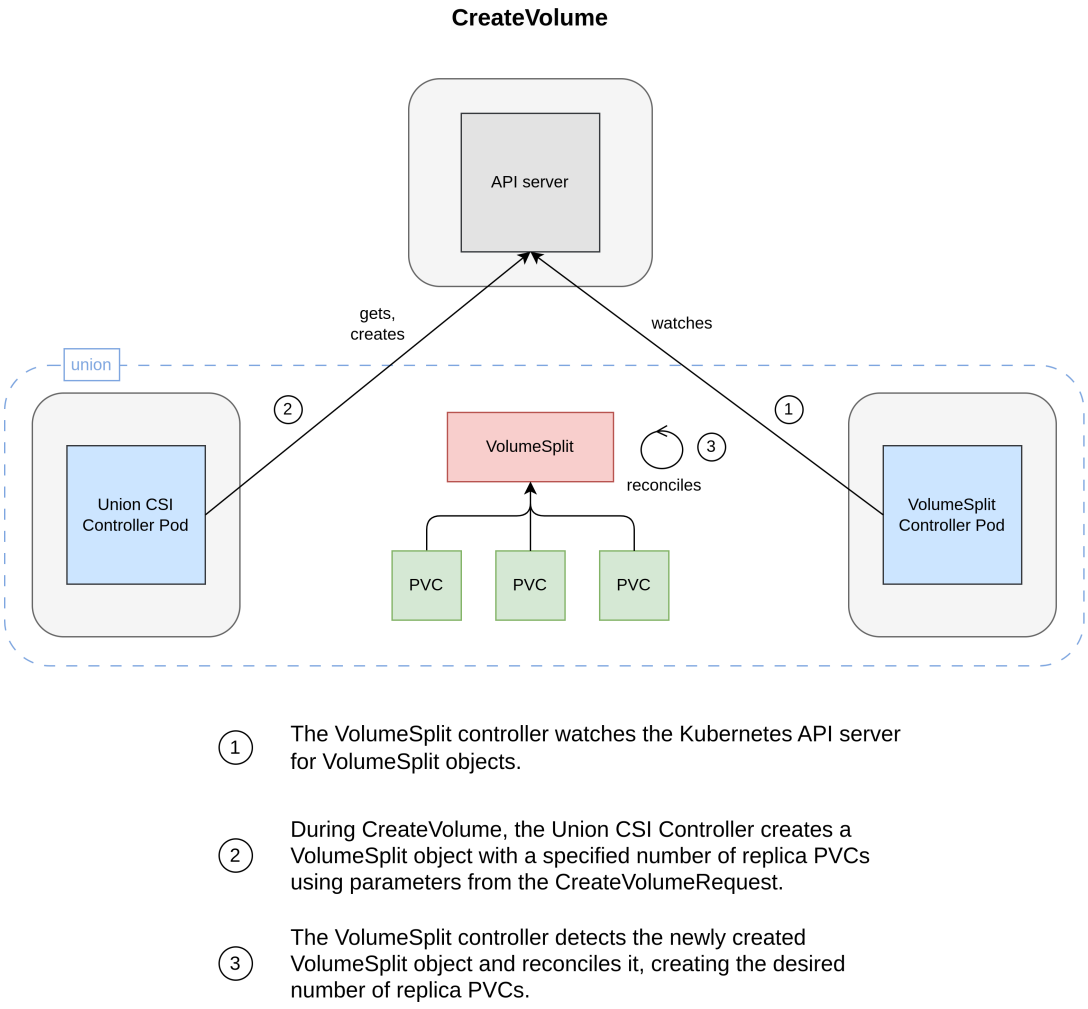


Figure 3.5: *How a VolumeSplit object and its replica PVCs are created*

Additionally, the controller observes the state of the VolumeSplit and its owned PVCs and updates the VolumeSplit `status`. The controller derives summary information regarding the provisioning state of the lower volumes by examining the status fields of the corresponding PVCs and conveys this information within the VolumeSplit's `conditions` array. The PersistentVolume objects of the lower volumes are beyond the controller's scope of control and are neither monitored nor referenced by the controller.

The reconciliation loop of the VolumeSplit controller regarding the management of the PVC replicas is straightforward:

1. The controller lists all the PVCs in the same namespace as the VolumeSplit object that have an owner reference to the VolumeSplit.
2. If the listed PVCs are fewer than `spec.replicas`:

   (a) The controller creates the required number of PVCs in the API server, using the VolumeSplit's `spec.template` and setting the VolumeSplit object as the owner of the PVCs with `blockOwnerDeletion` set to true.

3. If the listed PVCs are more than `spec.replicas`:

   (a) The controller does nothing and reports that scaling down functionality is not implemented. Selecting which surplus PVCs to delete is a difficult task, and in case all the PVCs are bound, it can result in data loss. Additionally, the Union CSI plugin does not currently need to scale down PVCs.

Given the logic described above, it is evident that the controller does not list all the existing replicas in the same namespace as the owner (regardless of owner references) and try to take ownership of orphaned replicas (replicas that do not have an owner reference but match the label selector) or release owned replicas that no longer match the label selector. This is the kind of replica reconciliation logic found in many Kubernetes built-in controllers that manage sets of replicas, like the ReplicaSet controller.

VolumeSplit objects are meant to be deleted by the *Union CSI Controller* during the `DeleteVolume` RPC. Foreground cascading deletion is preferred in order to block the removal of the VolumeSplit instance until all its owned PVCs are removed first.

We must point out that, even though the owned PVCs are removed before, after or regardless of the VolumeSplit, dictated by the cascading deletion types of foreground, background and orphan respectively, the corresponding PVs are not included in the cascading deletion. The PVs can only be removed when their underlying volumes are deleted. This process may take a long time, even forever if the corresponding volume plugin is experiencing internal errors or has been removed from the cluster. The VolumeSplit controller cares only for the VolumeSplit object and the owned PVCs being removed, and does so by using owner references.

The VolumeSplit controller is a separate binary from the Union CSI plugin and is meant to be deployed in its own container as a stand-alone Deployment in the `union` namespace.

3.5.13 Sense of Attachment

In the same spirit as Section 3.5.12, Union CSI must define what constitutes the attachment and detachment of its volumes, as well as how these actions are initiated, monitored, and verified.

A Union CSI volume consists of a number of lower volumes or branches, which are requested by Union CSI through PersistentVolumeClaims on the lower volume plugin. To make a Union CSI volume accessible on a node, every branch must be accessible on the same node, either locally or remotely through the network. In addition, the branches must be merged by a MergerFS process on the host, and the union filesystem must be mounted on the host filesystem in the staging area of *Union CSI Node* (`/var/lib/union-csi/`).

To avoid installing MergerFS directly on each host node and to conveniently manage the branches through their PVCs, a special privileged Pod handles the attachment process described above. This Pod uses the PVCs of the branches of the Union CSI volume on a container that wraps the MergerFS program and is assigned to the same node where the user's workload will run. MergerFS is executed within the container, and the merged filesystem is published on the host. This MergerFS filesystem on the host node constitutes the storage asset of a Union CSI volume, represented in the cluster by a VolumeSplit object.

Pod (or *Attach-Pod*...)

During `ControllerPublishVolume`, the *Union CSI Controller* must attach the specified volume to the specified node. The *Union CSI Controller* retrieves the VolumeSplit object corresponding to the volume identifier in the `ControllerPublishVolumeRequest`. Next, Union CSI creates a Pod that uses the PVC replicas of the VolumeSplit plus an additional `hostPath` volume to mount the filesystem on the host. These volumes are all mounted within a MergerFS container running in privileged mode (equivalent to root on the host). The Pod is created on the Kubernetes API in the same namespace as the PVCs and assigned to the specified node.

Since this Pod uses one or more volumes of the lower volume plugin and is scheduled on a node, Kubernetes triggers the attachment and mounting operations of these volumes on the same node by the lower plugin on behalf of Union CSI. Union CSI has only to wait

for the Pod's phase to transition to `Running` to ensure the lower volumes are successfully published on the node by the lower plugin, and the MergerFS filesystem is mounted on the host by the Pod container.

From now on, this particular Pod created by the Union CSI plugin will often be referred to as an *"Attach-Pod"*, since its creation kick-starts the attachment (and mounting) of the lower volumes. *"Attach-Pod"* is not a new, custom resource, just a handy term that can help distinguish between consumer Pods created by the user.

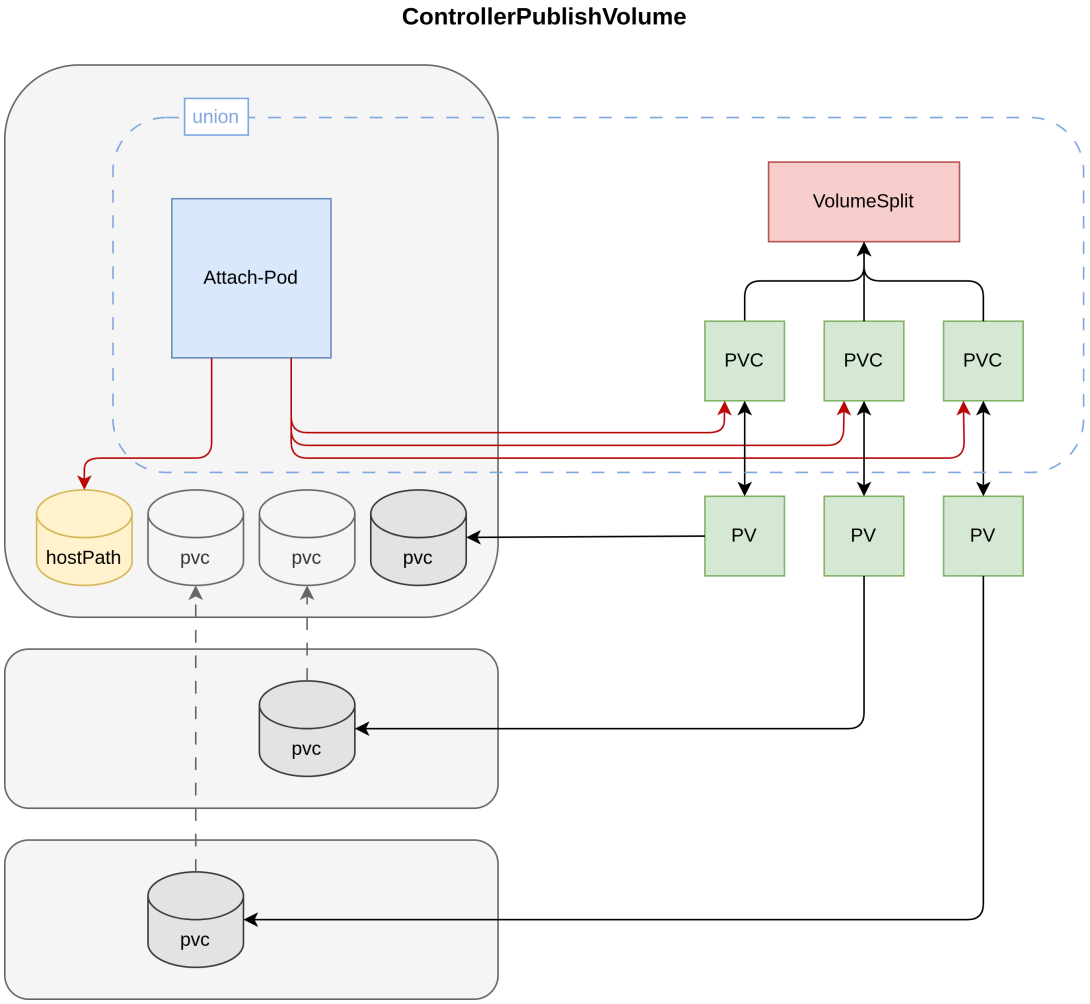


Figure 3.6: *An Attach-Pod using the PVCs of a VolumeSplit and a HostPath volume*

Listing 3.4 shows an example of an *Attach-Pod* for a volume with `volume_id` of `pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f` and node with `node_id` of `kind-worker2`.

Listing 3.4: *Example of an Attach-Pod*

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
```

```
  name: pod-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f
  namespace: union
spec:
  containers:
  - args:
    - -c
    - gogomergerfs mergerfs --branches=/volume/branches/\*
      --target=/volume/merged --block
    command:
    - /bin/sh
    image: docker.io/on2e/gogomergerfs:dev-mergerfs2.37.1
    imagePullPolicy: Always
    name: gogomergerfs
    securityContext:
      privileged: true
    volumeMounts:
    - mountPath: /volume/branches/branch0-split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-b22xk
      name: branch0
    - mountPath: /volume/branches/branch1-split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-h8rgj
      name: branch1
    - mountPath: /volume/merged
      mountPropagation: Bidirectional
      name: target
  nodeSelector:
    kubernetes.io/hostname: kind-worker2
  volumes:
  - name: branch0
    persistentVolumeClaim:
      claimName: split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-b22xk
  - name: branch1
    persistentVolumeClaim:
      claimName: split-pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f-h8rgj
  - hostPath:
      path: /var/lib/union-csi/volumes/pvc-7d4623bf-2832-4431-8c32-2bbdaca9e26f/merged
      type: DirectoryOrCreate
    name: target
```

The Pod in Listing 3.4 is named `pod-<volume_id>` in the `union` namespace, and includes:

- The following container:
  - **`gogomergerfs`**: The container executes the `mergerfs` command through another program called `gogomergerfs` and blocks. The container runs in privileged mode. This is necessary because MergerFS needs root privileges and access to the `/dev/fuse` character device on the host filesystem. Upon mounting the filesystem, the MergerFS FUSE daemon process is running on the host to proxy the user filesystem requests and the underlying branches. For more details about the `gogomergerfs` program, see the next paragraph and Section 4.2.1.
- A node selector using the following label:
  - **`kubernetes.io/hostname`**: The Pod is restricted to run on node `<node_id>` with a `nodeSelector` using the well-known label `kubernetes.io/hostname`. This label, reserved by Kubernetes, is placed by Kubelet on the Node objects of the cluster and is populated with the hostname of the node. Note that the hostname can be changed from the "actual" hostname by passing the `--hostname-override` flag to `kubelet`. As a result, this label may not always

hold the node identifier as returned by the `NodeGetInfo` RPC. Future versions of Union CSI may use a different, stable label, set by Union CSI itself.

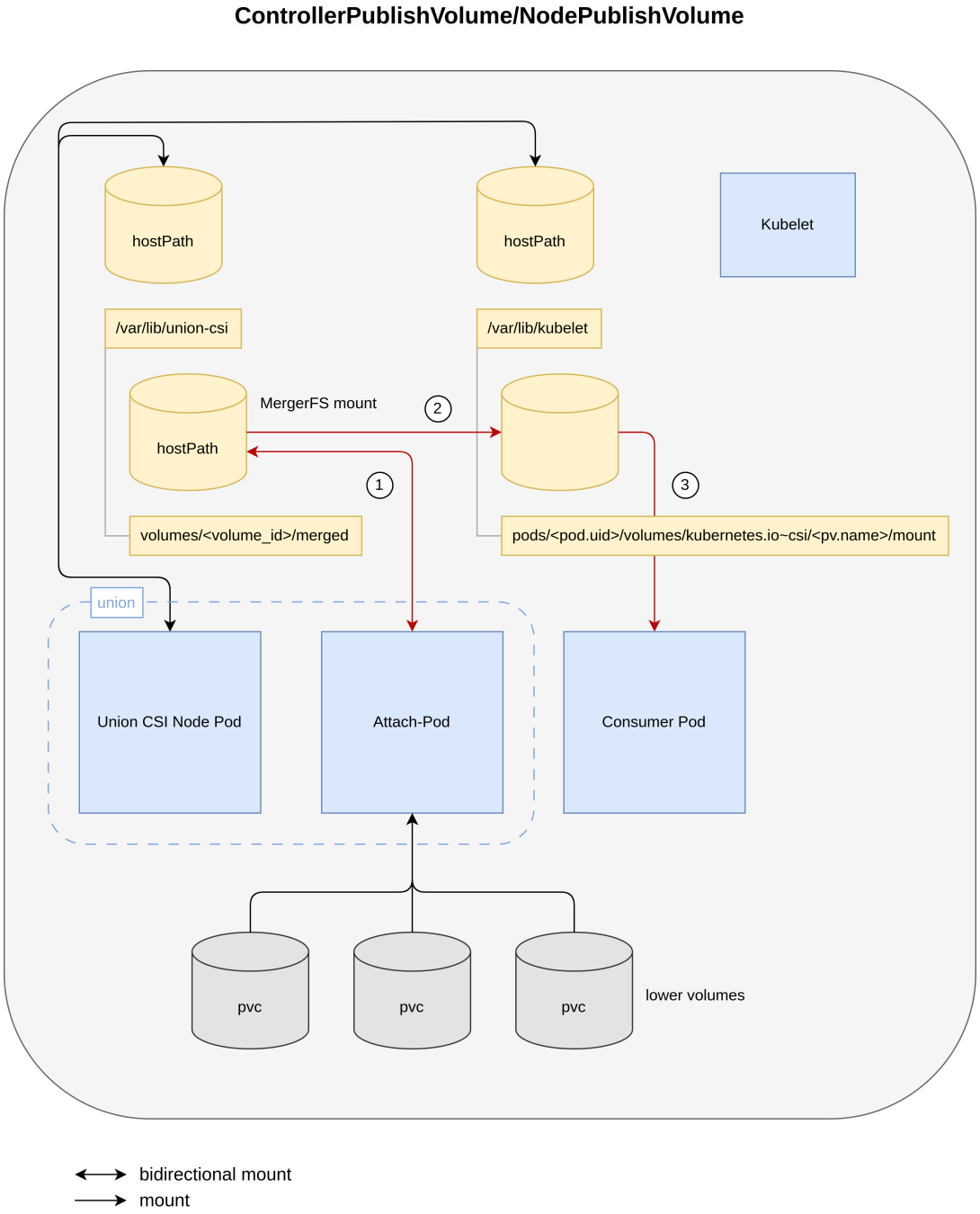
- The following volumes:
  - **`branch*`**: The `persistentVolumeClaim` volumes of the lower replica PVCs mounted on the container under `/volumes/branches/` at their respective directories. These volumes are the branches of the MergerFS filesystem.
  - **`target`**: The `hostPath` volume of `/var/lib/union-csi/volumes/<volume_id>/merged` mounted on the container at `/volumes/merged`. This is the mount point of the MergerFS filesystem. The volume is mounted with `Bidirectional` mount propagation so the mounted filesystem is propagated on the host under `/var/lib/union-csi/` and *Union CSI Node* can bind-mount it at the specified target path during `NodePublishVolume`.

Once an *Attach-Pod* successfully merges and mounts the Union CSI volume (the MergerFS filesystem) on the host, it continues running on the Node. The *Union CSI Controller* interprets the Pod's `status.phase` of `Running` and `spec.nodeName` of `<node_id>` as confirmation that the Union CSI volume is attached and staged on the specified Node.

Subsequently, during `ControllerUnpublishVolume`, the *Union CSI Controller* retrieves and deletes the corresponding *Attach-Pod* and waits for its removal from the API server as confirmation that the Union CSI volume is detached. When the *Attach-Pod* is deleted, the MergerFS filesystem gracefully unmounts from the host, and the branch volumes of the lower storage system undergo detachment from the node. When a user creates a new consumer Pod that uses the Union CSI volume, whether on the same or a different Node, the *Union CSI Controller* reinitiates the process. It recreates an *Attach-Pod*, utilizing the lower volumes to merge them again and mount the Union CSI volume on the specified node.

MergerFS container

Within the MergerFS container, each of the PVC (input) volumes are mounted under the `/volume/branches/` directory, and the `hostPath` (output) volume is mounted at `/volume/merged/`. The ”branches” parameter of `mergerfs` is set to `/volume/branches/*` to include all the directories under `/volume/branches/`, and the ”mountpoint”



1 During ControllerPublishVolume, the Attach-Pod mounts the MergerFS filesystem on the host under /var/lib/union-csi.

2 During NodePublishVolume, the Union CSI Node bind-mounts the MergerFS filesystem at the target path under /var/lib/kubelet.

3 Kubelet bind-mounts the target path into the consumer Pod container(s).

Figure 3.7: *How a Union CSI volume is mounted on a consumer Pod*

parameter is set to `/volume/merged`. Upon execution of `mergerfs`, the union filesystem is mounted at `/volume/merged` and propagated on the host at the *Union CSI Node* managed directory `/var/lib/union-csi/volumes/<volume_id>/merged`.

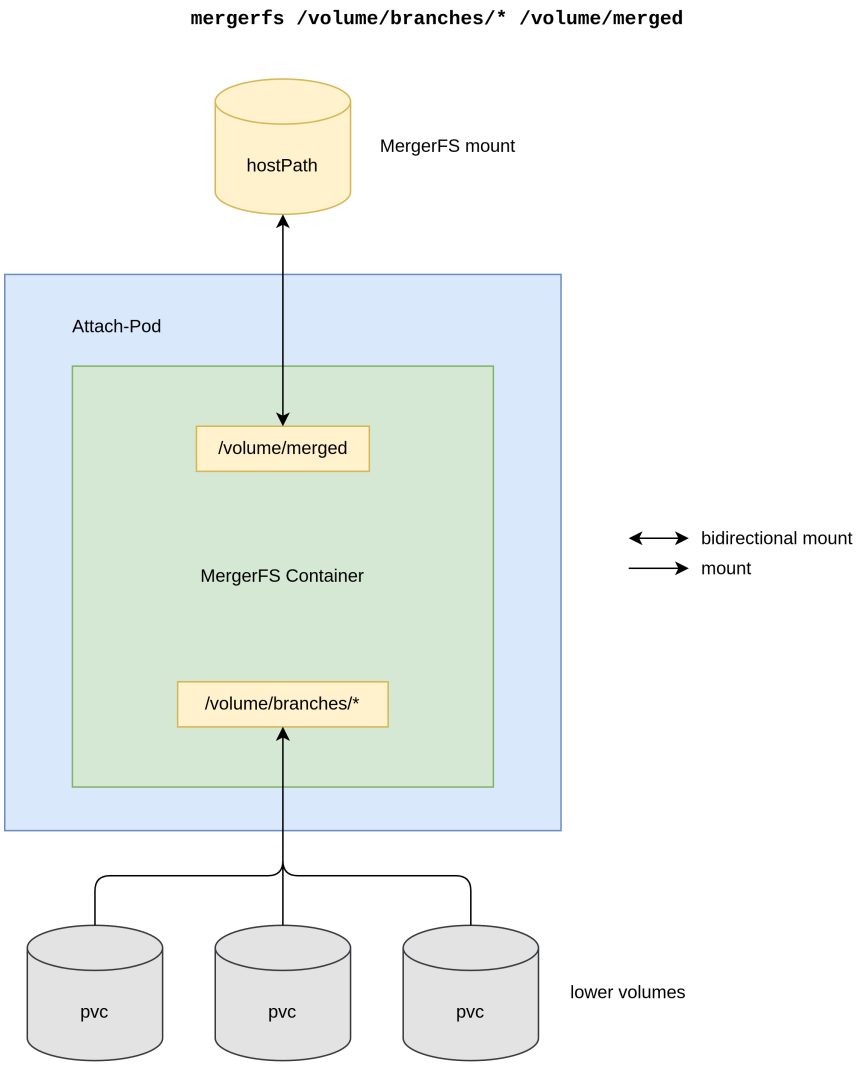


Figure 3.8: *The MergerFS container of an Attach-Pod and its mounted volumes*

After a successful merge, the *Attach-Pod* must remain running on the node to serve the MergerFS filesystem until it is deleted by Union CSI during the `ControllerUnpublishVolume` RPC. However, by default the successful execution of the `mergerfs` binary mounts the resulting union filesystem and exits; it does not block. The foreground `mergerfs` process is *daemonized*: the current process is forked and the parent process is killed. The forked process is the userspace FUSE handler of the filesystem that runs

in the background and processes the IO requests of the user (see Section 2.5.1). This means that if we were to run a container with `mergerfs` as the entry point, `mergerfs` would run inside the container, exit with 0 status on success, the FUSE handler and all container processes would terminate, and the container exit. A Pod that runs such a container would transition from `Running` to `Succeeded` and remain in this state until it is deleted or garbage collected. The MergerFS container of the *Attach-Pod* must block after a successful mount so the MergerFS FUSE handler can keep running.

Another problem occurs when a MergerFS container exits while the MergerFS filesystem is mounted on the host with shared/bidirectional mount propagation. If the container terminates before the filesystem is unmounted, the FUSE daemon process also terminates. Any mounts of the filesystem outside the container will now appear as corrupted. This is because the handler process that was registered with the kernel to forward the filesystem IO requests to, no longer exists. The kernel does not know how to respond to requests and how to interact with the filesystem. The corrupted mount points must be unmounted first before they are usable again. Listing 3.5 shows how a corrupted MergerFS mount is displayed on the terminal after killing the MergerFS handler and how the original directory is restored after unmounting the filesystem.

Listing 3.5: *The MergerFS filesystem appears corrupted if the FUSE handler is killed*

```
# ls -il

total 12
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
6291862 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint

# mergerfs branch1:branch2 mountpoint
# ls -il

total 12
            6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
            6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
5424983562661234939 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint

# ps aux | grep mergerfs

root     1718664  0.0  0.0 901132  5076 ?        S<sl 23:01   0:00 mergerfs branch1:branch2 ↩
    mountpoint
root     1718815  0.0  0.0   9088  2432 pts/7    S+   23:01   0:00 grep --color=auto mergerfs

# kill -9 1718664
# ls -il

ls: cannot access 'mountpoint': Transport endpoint is not connected
total 8
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
      ? d????????? ? ?    ?       ?            ? mountpoint

# touch mountpoint/file
touch: cannot touch 'mountpoint/file': Transport endpoint is not connected

# umount mountpoint
# ls -il

total 12
6291708 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch1
6291791 drwxr-xr-x 2 root root 4096 Nov 20 23:00 branch2
```

```
39 6291862 drwxr-xr-x 2 root root 4096 Nov 20 23:00 mountpoint
```

Union CSI volumes are MergerFS filesystems mounted on a node through bidirectional mount propagation from *Attach-Pod* MergerFS containers. If a container terminates without unmounting the filesystem first, the host directory `/var/lib/union-csi/volumes/<volume_id>/merged` will be damaged. In such a scenario, the directory will have to be manually unmounted by the *Union CSI Node* on the node. Otherwise, new user Pods attempting to use this volume on the same node will fail because the now corrupted `hostPath` volume cannot be mounted by Kubelet on a new *Attach-Pod*. Hence, it is crucial for the MergerFS container of the *Attach-Pod* to unmount the filesystem from the container directory `/volume/merged` (and thus from the corresponding host directory) when the Pod is terminating.

To address these requirements, we wrote a Go wrapper program for the `mergerfs` command. The `gogomergerfs` program, as seen in the example Pod in Listing 3.4, executes the `mergerfs` command, waits for UNIX signals (`SIGTERM` and `SIGINT`), and proceeds to unmount the filesystem from the target directory before exiting. For more details about the program and its implementation visit Section 4.2.1.

3.5.14 Sidecar Containers

As described in Section 3.5.8, the Union CSI driver implements the following RPCs that involve volume operations:

- **`CreateVolume`**
- **`DeleteVolume`**
- **`ControllerPublishVolume`**
- **`ControllerUnpublishVolume`**
- **`NodePublishVolume`**
- **`NodeUnpublishVolume`**

Consequently, the Kubernetes sidecar containers necessary by and deployed with the Union CSI driver are:

- **`external-provisioner`** (invokes `CreateVolume`/`DeleteVolume`)

- **`external-attacher`** (invokes `ControllerPublishVolume`/ `ControllerUnpublishVolume`)
- **`node-driver-registrar`** (registers plugin with Kubelet which invokes `NodePublishVolume/NodeUnpublishVolume`)

Refer to Sections 2.4.1 and 2.4.2 for more information regarding the above sidecar containers.

3.5.15 Communication

The CO communicates with a CSI plugin through gRPC in order to access its provided services (Identity, Controller and Node). This communication takes place over UNIX domain sockets.

The UNIX domain socket endpoint is provided to a Union CSI plugin instance by the cluster administrator through the `CSI_ENDPOINT` environmental variable (fo example, `CSI_ENDPOINT=unix:///csi.sock`) and any related command-line options it might implement. The same endpoint is also provided to any sidecar containers accompanying the plugin master container in the same Pod. The Union CSI driver creates, binds, and listens to the specified UNIX socket. Subsequently, the sidecar containers or Kubelet probe the endpoint and invoke the necessary gRPC calls of the driver.

3.6 Example Walkthrough

In this section, we present a concise workflow of the Union CSI's main volume operations, based on the analysis provided in the previous sections and following the general flow of success.

3.6.1 Creating Volumes

1. A cluster administrator creates a StorageClass objects referring to the Union CSI driver and a StorageClass referring to a lower volume plugin.
2. A user creates a PersistentVolumeClaim, in his namespace of choice, referring to the Union CSI StorageClass.

3. The `external-provisioner` sidecar container of the Union CSI driver detects the unbound PVC and invokes the `CreateVolume` RPC of the driver.
4. If `CreateVolume` returns successfully, the `external-provisioner` creates a PersistentVolume object to represent the newly provisioned volume and binds it to the PersistentVolumeClaim.

The `CreateVolume` RPC of Union CSI is shown in Algorithm 9.

Algorithm 9: `CreateVolume` RPC of the Union CSI plugin (high-level)

request : `CreateVolumeRequest`

response: `CreateVolumeResponse, error`

1. **if** `CreateVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `CreateVolumeRequest.name`
   `capacity` ← `CreateVolumeRequest.capacity_range.required_bytes`
   `class` ← `CreateVolumeRequest.parameters["lowerStorageClassName"]`
   `rwo` ← `ReadWriteOnce`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit exists* **then**
   compare the VolumeSplit parameters with those from the `CreateVolumeRequest` (idempotency):
   if `spec.capacity` < `capacity` **or**
   `spec.template.spec.storageClassName` ≠ `class` **or**
   `spec.template.spec.accessModes` ≠ `[rwo]`
   then return with `ALREADY_EXISTS` error code and a related message.
   else
   create a new VolumeSplit `union/split-<volumeID>`. The VolumeSplit fields are populated using values from the `CreateVolumeRequest`, as described in Algorithm 8. The VolumeSplit controller creates the desired replica PVCs in the same namespace as the VolumeSplit.
5. Poll the VolumeSplit object using exponential backoff for a short period waiting for one of the following conditions:
   (a) **if** *VolumeSplit* `status` *contains a condition type of* `Ready` *or* `Pending` *with condition status of* `True` **then** polling succeeds.
   (b) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.
6. **if** *polling succeeds* **then**
   `CreateVolumeResponse.volume.volume_id` ← `volumeID`
   `error` ← `nil`

3.6.2 Deleting Volumes

1. A user deletes the PVC bound to the PV backed by the Union CSI volume.
2. The `external-provisioner` of the Union CSI driver detects the deletion of the PVC and invokes the `DeleteVolume` RPC of the driver.
3. If `DeleteVolume` returns successfully, the `external-provisioner` deletes the released PV object.

The `DeleteVolume` RPC of Union CSI is shown in Algorithm 10.

Algorithm 10: `DeleteVolume` RPC of the Union CSI plugin (high-level)

request : `DeleteVolumeRequest`

response: `DeleteVolumeResponse, error`

1. **if** `DeleteVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `DeleteVolumeRequest.volume_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `OK` error code (idempotency) .
5. **if** `deletionTimestamp` $\neq$ `nil` **then** delete the VolumeSplit using foreground cascading deletion (so the owned PVCs are removed before the VolumeSplit) .
6. Poll the VolumeSplit object using exponential backoff for a short period waiting for one of the following conditions:
   (a) **if** *VolumeSplit is removed from the API server* **then** polling succeeds.
   (b) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

   The `DeleteVolume` request may time out during this period.
7. **if** *polling succeeds* **then**
   `DeleteVolumeResponse` ← `{}`
   `error` ← `nil`

3.6.3 Attaching Volumes

1. A user creates a Pod that uses the PVC referring to the Union CSI volume.
2. The scheduler selects a Node for the Pod and the in-tree CSI volume plugin creates a VolumeAttachment object.

3. The `external-attacher` sidecar container of the Union CSI driver detects the VolumeAttachment and invokes the `ControllerPublishVolume` RPC of the driver.
4. If `ControllerPublishVolume` returns successfully, the `external-attacher` updates the VolumeAttachment object to indicate the successful attachment of the volume.

The `ControllerPublishVolume` RPC of Union CSI is shown in Algorithm 11.

Algorithm 11: `ControllerPublishVolume` RPC of the Union CSI plugin (high-level)

request : `ControllerPublishVolumeRequest`

response: `ControllerPublishVolumeResponse, error`

1. **if** `ControllerPublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `ControllerPublishVolumeRequest.volume_id`
   `nodeID` ← `ControllerPublishVolumeRequest.node_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
5. Get Node object `<nodeID>`.
6. **if** *Node does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
7. Get Pod object `union/pod-<volumeID>`.
8. **if** *Pod does NOT exist* **then** create a new Pod `union/pod-<volumeID>`. The Pod carries the MergerFS container, uses the PVCs of the VolumeSplit and a `hostPath` volume, and is assigned to Node `<nodeID>`, as described in Section 3.5.13. The Pod merges the PVC volumes and mounts the MergerFS filesystem on the `hostPath` volume at **"**`/var/lib/union-csi/volumes/<volumeID>/merged/`**"** on the host .
9. Poll the Pod using exponential backoff for a short period waiting for one of the following conditions:
   (a) **if** *Pod is terminating (has phase of* `Failed` *or* `Succeeded`*)* **then return** with `INTERNAL` error code and a related message.
   (b) **if** *Pod is running on a Node that is not* `<nodeID>` **then return** with `FAILED_PRECONDITION` error code and a related message.
   (c) **if** *Pod is running on Node* `<nodeID>` **then** polling succeeds.
   (d) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

   The `ControllerPublishVolume` request may time out during this period.
10. **if** *polling succeeds* **then**
    `ControllerPublishVolumeResponse` ← `{}`
    `error` ← `nil`

3.6.4 Detaching Volumes

1. A user deletes the Pod that refers to the Union CSI volume and is scheduled on a Node.
2. The in-tree CSI volume plugin deletes the corresponding VolumeAttachment object that is protected by a finalizer.
3. The `external-attacher` of the Union CSI driver detects the `deletionTimestamp` on the VolumeAttachment and invokes the `ControllerUnpublishVolume` RPC of the driver.
4. If `ControllerUnpublishVolume` returns successfully, the `external-attacher` removes the finalizer from the VolumeAttachment to allow for it to be removed from the API and indicate the successful detachment of the volume.

The `ControllerUnpublishVolume` RPC of Union CSI is shown in Algorithm 12.

Algorithm 12: `ControllerUnpublishVolume` RPC of the Union CSI plugin (high-level)

request : `ControllerUnpublishVolumeRequest`

response: `ControllerUnpublishVolumeResponse, error`

1. **if** `ControllerUnpublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `ControllerUnpublishVolumeRequest.volume_id`
   `nodeID` ← `ControllerUnpublishVolumeRequest.node_id`
3. Get VolumeSplit object `union/split-<volumeID>`.
4. **if** *VolumeSplit does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
5. Get Node object `<nodeID>`.
6. **if** *Node does NOT exist* **then return** with `NOT_FOUND` error code and a related message .
7. Get Pod object `union/pod-<volumeID>`.
8. **if** *Pod does NOT exist* **then return** with `OK` error code (idempotency).
9. **if** *Pod is scheduled on a Node that is not* `<nodeID>` *or is not yet scheduled* **then return** with `OK` error code (idempotency).
10. **if** *Pod is scheduled on Node* `<nodeID>` *and* `deletionTimestamp` $\neq$ `nil` **then** delete the Pod.
11. Poll the Pod using exponential backoff for a short period waiting for one of the following conditions:
    (a) **if** *Pod is removed from the API server* **then** polling succeeds.
    (b) **if** *polling times out* **or** *RPC request times out* **then return** with `DEADLINE_EXCEEDED` error code and a related message.

    The `ControllerUnpublishVolume` request may time out during this period.
12. **if** *polling succeeds* **then**
    `ControllerUnpublishVolumeResponse` ← `{}`
    `error` ← `nil`

3.6.5 Mounting Volumes

1. Kubelet detects that the Pod referring to a Union CSI volume has been scheduled on the Node and invokes the `NodePublishVolume` RPC of the Union CSI driver via the registered UNIX domain socket.
2. If `NodePublishVolume` returns successfully, the specified target path where the Union CSI volume is mounted is further mounted by Kubelet on the Pod container.

The `NodePublishVolume` RPC of Union CSI is shown in Algorithm 13.

Algorithm 13: `NodePublishVolume` RPC of the Union CSI plugin (high-level)

request : `NodePublishVolumeRequest`

response: `NodePublishVolumeResponse, error`

1. **if** `NodePublishVolumeRequest` *is missing required fields or has invalid values* **then return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `NodePublishVolumeRequest.volume_id`
   `targetPath` ← `NodePublishVolumeRequest.target_path`
   `sourcePath` ← ”`/var/lib/union-csi/volumes/<volumeID>/merged/`”
3. **if** `<targetPath>` *does NOT exist on the host or is corrupted* **then return** with `INTERNAL` error code and a related message .
4. **if** `<sourcePath>` *does NOT exist on the host or is corrupted* **then return** with `INTERNAL` error code and a related message .
5. **if** `<sourcePath>` *is ALREADY mounted on* `<targetPath>` **then return** with `OK` error code (idempotency).
6. Mount (bind) `<sourcePath>` on `<targetPath>`.
7. **if** *mounting succeeds* **then**
   `NodePublishVolumeResponse` ← `{}`
   `error` ← `nil`

3.6.6 Unmounting Volumes

1. Kubelet detects that the Pod referring to the Union CSI volume is terminated or deleted on the Node and invokes the `NodeUnpublishVolume` RPC of the Union CSI driver via the registered UNIX domain socket.

2. If `NodeUnpublishVolume` returns successfully, the specified target path where the Union CSI volume was mounted is unmounted from the Pod container.

The `NodeUnpublishVolume` RPC of Union CSI is shown in Algorithm 14.

Algorithm 14: `NodeUnpublishVolume` RPC of the Union CSI plugin (high-level)

request : `NodeUnpublishVolumeRequest`

response: `NodeUnpublishVolumeResponse, error`

1. **if** `NodeUnpublishVolumeRequest` *is missing required fields or has invalid values* **then** **return** with `INVALID_ARGUMENT` error code and a related message.
2. `volumeID` ← `NodeUnpublishVolumeRequest.volume_id`
   `targetPath` ← `NodeUnpublishVolumeRequest.target_path`
3. **if** `<targetPath>` *does NOT exist on the host or is corrupted* **then** **return** with `INTERNAL` error code and a related message .
4. **if** `<targetPath>` *is not mounted by a filesystem* **then return** with `OK` error code (idempotency).
5. Unmount `<targetPath>`.
6. **if** *unmounting succeeds* **then**
   - `NodeUnpublishVolumeResponse` ← `{}`
   - `error` ← `nil`

4 Implementation

In this chapter, we provide a concise description of the implementation and deployment details of the Union CSI application.

4.1 Overview

The implementation of the Union CSI application comprises three distinct components:

- **Gogomergerfs:** The program that wraps and executes the `mergerfs` command.
- **VolumeSplit Controller:** The VolumeSplit CRD and controller.
- **Union CSI:** The CSI driver and storage backend.

Each component is written in the Go programming language and its code resides in the same umbrella repository, called "Union CSI", within its dedicated directory. Additionally, every component is compiled into an individual binary and Docker image, utilizing its own Makefile and Dockerfile. All components share a common flat (non-semantic) software version called "dev".

The Gogomergerfs container is used and deployed by the Union CSI plugin everytime it creates a Pod (*Attach-Pod*) in Kubernetes to merge the lower volumes.

Both the VolumeSplit controller and Union CSI containers are deployed in a Kubernetes cluster using YAML manifests. *Kustomization* files (`kustomization.yaml`) facilitate installing and uninstalling the entire application and its resources in Kubernetes with a simple `kubectl apply -k` or `delete -k` command.

Union CSI ”dev” is compatible with:

- **Kubernetes:** v1.26+
- **CSI specification:** v1.8.0
- **MergerFS:** v2.37.0
- **OS:** Linux

4.2 Components

In this section, we delve into the implementation of the key components of the Union CSI application, offering insights into their internal architecture and providing an overview of their source code.

4.2.1 Gogomergerfs

Gogomergerfs is a Go wrapper program for the `mergerfs` command. It is designed to be employed by the Union CSI plugin to combine volumes into a MergerFS filesystem from within a Pod and offer specific *quality of life* functionalities after the filesystem is mounted.

The main subcommand of `gogomergerfs` is `mergerfs`, which executes the `mergerfs` program. The available options for this subcommand are shown in Table 4.1.

Option	Values	Default	Description
`--branches`			Comma-separated list of paths to merge together.
`--target`			The union mount point.
`-o, --options`			Comma-separated list of mount options.
`--block`	true, false	false	Execute `mergerfs`, block for `SIGINT` or `SIGTERM`, then unmount. If set to false, execute `mergerfs` as if executing directly the command.

Table 4.1: *”gogomergerfs mergerfs” command options*

The `gogomergerfs mergerfs` command tries to mimic the command-line syntax of the original `mergerfs`. The command

```
gogomergerfs mergerfs -o=<o1>,<o2> --branches=b1,b2,b3 --target=t
```

invokes

```
mergerfs -o <o1>,<o2> b1:b2:b3 t
```

maintaining the same behavior as executing `mergerfs` directly.

If the `--block` argument is given, `gogomergerfs` executes `mergerfs` and then blocks, waiting for the `SIGINT` or `SIGTERM` signals. When such a signal is received, `gogomergerfs` attempts to unmount the MergerFS filesystem from the target mount point before exiting.

With Gogomergerfs, the lifecycle of the MergerFS filesystem (the Union CSI volume) aligns with the Pod's intended lifecycle. The filesystem remains active while the Pod is running and is unmounted from the host when the container receives a `SIGTERM` to terminate. The Gogomergerfs wrapper program can be extended with additional features, such as monitoring the mounted filesystem and logging about its status while on hold for the signals, facilitating debugging. For more information on how `SIGTERM` is sent to Pod containers and the grace period they are given for clean up operations, see Section 2.2.1.5.

To run `mergerfs` directly or through `gogomergerfs` within a Docker container, the `--cap-add=SYS_ADMIN --device=/dev/fuse --security-opt=apparmor:unconfined` options should be included. Additionally, for the MergerFS filesystem to propagate from the container to the host, the target directory should be mounted with `rshared` mount propagation [1].

For `gogomergerfs` to receive UNIX signals when running inside a container, it must be the PID 1 process of the container. For Docker images, the *exec* form of the `ENTRYPOINT`, such as `ENTRYPOINT ["gogomergerfs", "mergerfs", ...]`, would be suitable for this purpose. However, there are instances where running `mergerfs` within a shell is preferred to leverage shell processing features like environment variable substitution and globbing for branch and mount point paths. In such cases, either the *shell* form or the *exec* form with a direct shell invocation can be used. In both forms, starting

[1] https://github.com/trapexit/mergerfs#can-mergerfs-run-via-docker-podman-kubernetes-etc

with `exec` ensures that `gogomergerfs` will be the PID 1 process of the container and not the shell. For the *shell* form:

```
ENTRYPOINT exec gogomergerfs mergerfs ...
```

For the *exec* form:

```
ENTRYPOINT ["/bin/sh", "-c", "exec gogomergerfs mergerfs ..."]
```

The Dockerfile of the Gogomergerfs component builds into the same image both the `gogomergerfs` and `mergerfs` binaries, enabling the former to execute the latter when the container runs.

The key points of Gogomergerfs's implementation are exposed in the following Listings.

Listing 4.1: *The Merger and BlockingMerger interfaces*

```
// Merger defines an interface that abstracts over union mount implementations.
type Merger interface {
    // Merge combines the contents of branches using options and serves the union mount at target.
    Merge(branches []string, target string, options []string) error
    // Unmerge undoes the result of Merge, e.g. unmounts the union mount from target.
    Unmerge(target string) error
}

// BlockingMerger is a Merger that can be used to block after a successful Merge
// and perform a clean up on the union mount when stopped.
type BlockingMerger interface {
    Merger
    // Run calls Merge and blocks until the context is cancelled
    // or Stop is called, whichever happens first.
    Run(ctx context.Context) error
    // Stop unblocks if running, returns without doing anything if not.
    // It is safe to call Stop multiple times.
    Stop()
    // CleanUp performs a clean up operation (e.g. unmount).
    // Returns without doing anything if running.
    // Meant to be used after successfully run and stopped.
    CleanUp() error
}

// NewBlockingMerger returns a new BlockingMerger.
func NewBlockingMerger(merger Merger, branches []string, target string, options []string) BlockingMerger {
    return &blockingMerger{
        merger:     merger,
        branches:   branches,
        target:     target,
        options:    options,
        logger:     log.New(os.Stderr, "", log.Ldate|log.Ltime|log.LUTC|log.Lshortfile|log.Lmsgprefix),
        manualStop: make(chan struct{}, 1),
        stopped:    make(chan struct{}, 1),
    }
}
```

Listing 4.2: *The BlockingMerger implementation*

```
// blockingMerger implements BlockingMerger.
type blockingMerger struct {
    mu *sync.Mutex
    // merger is the Merger implementation to use
    merger Merger
    // Arguments for Merge/Unmerge to store and use in Run
    branches []string
    target   string
    options  []string
    // logger is the logger to use
    logger *log.Logger
    // Channels for manual stop
    manualStop chan struct{}
```

```
    stopped    chan struct{}

  running bool
}

var _ BlockingMerger = &blockingMerger{}

func (bm *blockingMerger) Merge(branches []string, target string, options []string) error {
    bm.logger.Printf("Merging branches %q at target %q ...", branches, target)
    if err := bm.merger.Merge(branches, target, options); err != nil {
        return fmt.Errorf("failed to merge: %v", err)
    }
    bm.logger.Printf("Merged branches %q at target %q", branches, target)
    return nil
}

func (bm *blockingMerger) Unmerge(target string) error {
    bm.logger.Printf("Unmerging target %q ...", target)
    if err := bm.merger.Unmerge(target); err != nil {
        return fmt.Errorf("failed to unmerge: %v", err)
    }
    bm.logger.Printf("Unmerged target %q", target)
    return nil
}

func (bm *blockingMerger) Run(ctx context.Context) error {
    bm.mu.Lock()
    if bm.running {
        bm.mu.Unlock()
        return nil
    }

    if err := bm.Merge(bm.branches, bm.target, bm.options); err != nil {
        bm.mu.Unlock()
        return err
    }

    bm.running = true
    bm.mu.Unlock()
    stopRunning := func() {
        bm.mu.Lock()
        defer bm.mu.Unlock()
        bm.running = false
    }

    select {
    case <-ctx.Done():
        stopRunning()
    case <-bm.manualStop:
        stopRunning()
        bm.stopped <- struct{}{} // confirm stopped
    }

    return nil
}

func (bm *blockingMerger) Stop() {
    bm.mu.Lock()
    if !bm.running {
        bm.mu.Unlock()
        return
    }
    bm.manualStop <- struct{}{}
    bm.mu.Unlock()
    <-bm.stopped // wait stopped
}

func (bm *blockingMerger) CleanUp() error {
    bm.mu.Lock()
    defer bm.mu.Unlock()
    if bm.running {
        return nil
    }
    return bm.Unmerge(bm.target)
}
```

Listing 4.3: *mergerfs implements the Merger interface*

```
// mergerfs wraps the mergerfs union filesystem (https://github.com/trapexit/mergerfs),
// implementing the pkg/merger.Merger interface.
type mergerfs struct {
    binaryPath string
    mounter    mountutils.Interface
}

```

```
var _ merger.Merger = &mergerfs{}

func NewMergerfs() *mergerfs {
    return &mergerfs{
        binaryPath: "mergerfs",
        mounter:    mountutils.New(""),
    }
}

func (m *mergerfs) Merge(branches []string, target string, options []string) error {
    mergeCmd := m.binaryPath
    mergeArgs := []string{
        strings.Join(branches, ":"),
        target,
    }
    if len(options) > 0 {
        mergeArgs = append(mergeArgs, "-o", strings.Join(options, ","))
    }
    cmd := exec.Command(mergeCmd, mergeArgs...)
    output, err := cmd.CombinedOutput()
    if len(output) > 0 {
        fmt.Print(output)
    }
    if err != nil {
        // See k/k issue #103753
        if err.Error() == "wait: no child processes" {
            if cmd.ProcessState.Success() {
                return nil
            }
            err = &exec.ExitError{ProcessState: cmd.ProcessState}
        }
    }
    return err
}

func (m *mergerfs) Unmerge(target string) error {
    err := m.mounter.Unmount(target)
    // Ignore "not mounted" errors
    if err != nil && strings.Contains(err.Error(), "not mounted") {
        err = nil
    }
    return err
}
```

Listing 4.4: *The "gogomergerfs mergerfs" command*

```
type flags struct {
    Branches []string
    Target   string
    Options  []string
    Block    bool
}

func NewCommand() *cobra.Command {
    flags := &flags{}
    cmd := &cobra.Command{
        Args:  cobra.NoArgs,
        Use:   "mergerfs",
        Short: "Use mergerfs",
        Long:  "A featureful FUSE based union filesystem (https://github.com/trapexit/mergerfs↩
            )",
        RunE: func(cmd *cobra.Command, args []string) error {
            return runCommand(cmd, flags)
        },
    }
    cmd.Flags().StringSliceVar(
        &flags.Branches,
        "branches",
        []string{},
        "Comma-separated list of paths to merge together",
    )
    cmd.Flags().StringVar(
        &flags.Target,
        "target",
        "",
        "The union mount point",
    )
    cmd.Flags().StringSliceVarP(
        &flags.Options,
        "options",
        "o",
        []string{},
        "Comma-separated list of mount options",
    )
    cmd.Flags().BoolVar(
```

```
        &flags.Block,
        "block",
        false,
        "Execute mergerfs, block for SIGINT or SIGTERM, then unmount. If set to false, execute mergerfs as if executing directly the command",
    )
    cmd.Flags().SortFlags = false
    return cmd
}

func runCommand(cmd *cobra.Command, flags *flags) error {
    var mfs merger.Merger = mergerfs.NewMergerfs()

    if !flags.Block {
        return mfs.Merge(flags.Branches, flags.Target, flags.Options)
    }

    bm := merger.NewBlockingMerger(mfs, flags.Branches, flags.Target, flags.Options)
    if err := bm.Run(signal.SetupSignalHandler()); err != nil {
        return err
    }
    if err := bm.CleanUp(); err != nil {
        return err
    }

    return nil
}
```

Listing 4.5: *SIGINT and SIGTERM signals registration*

```
/*
    The contents of this file are lifted from sigs.k8s.io/controller-runtime/pkg/manager/signals
*/

var onlyOneSignalHandler = make(chan struct{})

// SetupSignalHandler registers for SIGTERM and SIGINT. A context is returned
// which is canceled on one of these signals. If a second signal is caught, the program
// is terminated with exit code 1.
func SetupSignalHandler() context.Context {
    close(onlyOneSignalHandler) // panics when called twice

    ctx, cancel := context.WithCancel(context.Background())

    c := make(chan os.Signal, 2)
    signal.Notify(c, syscall.SIGINT, syscall.SIGTERM)
    go func() {
        <-c
        cancel()
        <-c
        os.Exit(1) // second signal. Exit directly.
    }()

    return ctx
}
```

4.2.2 VolumeSplit Controller

The VolumeSplit Controller component is notably comprised of the VolumeSplit custom resource type definition in Go, its CRD YAML manifest, and a custom controller that handles the VolumeSplit API objects.

Kubebuilder SDK

This component was developed with the help of the *Kubebuilder* tool. Kubebuilder is a framework that significantly aids developers in building and publishing Kubernetes

APIs through CRDs, using Go and the command-line. Its powerful library offers a simple interface that builds on top and abstracts away the canonical way of writing Kubernetes APIs and controllers—a challenging, tedious and error-prone task, even for experienced Kubernetes developers. This enables users to quickly and efficiently customize Kubernetes.

Additionally, Kubebuilder's command-line tool automates various tasks, including:

- Creating a new project directory with a Makefile, a Dockerfile, and a suggested directory structure.
- Generating a new API with a given version, group, and kind, along with all necessary boilerplate. The developer can start filling out the API object's `spec` and `status` fields in Go.
- Generating the CRD YAML from the edited API definition and installing/uninstalling the CRD in the cluster.
- Updating the CRD file and boilerplate code around the API after editing.
- Generating scaffold code for the API controller that the developer can fill out with the reconciliation logic.
- Compiling, running locally, building into an image, pushing, deploying, and removing the controller using the provided Makefile and Dockerfile.

We utilized Kubebuilder to develop the VolumeSplit API, as showcased and described in Section 3.5.12.

The `Reconcile` method

Regarding the VolumeSplit controller, by utilizing the `controller-runtime` library of Kubebuilder the reconciliation logic is condensed to a single function, called `Reconcile`. This is where the developer adds his logic. The `Reconcile` function receives the name and namespace (for namespaced resources) of the object being reconciled, fetches the object from the local cache or the API, implements the control logic and returns an error, if the object needs to be requeued for reconciliation.

`Reconcile` is a method of a *reconciler*. A reconciler object registers with a higher-level object, called a `Manager`, that is provided by the `controller-runtime` library. The `Manager` is used to register and manage many different reconcilers, initializing their caches,

clients, queues and worker goroutines, and making the `Reconcile` callbacks. A running `Manager` watches the API server for create, update, and delete events regarding registered API resources, and invokes the corresponding `Reconcile` method for the changed object. A reconciler can inform the `Manager` for any API resources that it owns (through `metadata.ownerReferences`) so that `Reconcile` will be called on the owner object for any changes on the owned objects.

The reconciler struct for the VolumeSplit API is called `VolumeSplitReconciler`. The `main` function of the VolumeSplit Controller component creates a new `Manager` object, registers a `VolumeSplitReconciler` object with it, and starts the `Manager`.

The `Reconcile` method of the `VolumeSplitReconciler` fetches the VolumeSplit object (for which `Reconcile` was called by the `Manager`) and all the PVCs that are owned by it. If the current number of replica PVCs is lower than the desired number specified in `spec.replicas` of the VolumeSplit, `Reconcile` creates the required number of PVCs using the `spec.template` of the VolumeSplit.

More specifically, `Reconcile` executes the following steps:

1. Get the VolumeSplit instance being reconciled from the `Manager`'s cache using the given name and namespace.
2. If the VolumeSplit instance is not found in the API server, then it is considered removed and `Reconcile` returns with a nil error.
3. If the VolumeSplit instance exists, check if it is marked for deletion (`deletionTimestamp` is not nil).
   (a) If marked for deletion, the owner (VolumeSplit) and owned (PersistentVolumeClaim) objects are to be removed according to the specified cascading deletion type. Foreground cascading deletion is recommended for the delayed removal of the VolumeSplit until all its owned PVCs are removed first. Return from `Reconcile` to stop the reconciliation loop and let Kubernetes handle the garbage collection.
   (b) If not marked for deletion, proceed to the next steps.
4. List the PVCs in the same namespace as the VolumeSplit that have an `ownerReferences` field pointing to it.
5. Compare the number of listed PVCs with the desired number specified in `spec.replicas`

of the VolumeSplit.

(a) If PVCs are fewer than the desired number, create them in the API server in the same namespace as the VolumeSplit, with the VolumeSplit name as a prefix name for the PVCs (populating the `generateName field`), an `ownerReferences` field referring to the VolumeSplit (with `blockOwnerDeletion` set to true), and using `spec.template` for the `metadata` and `spec` fields of the PVCs.

(b) If PVCs are more than the desired number, log that scale down logic is not implemented and return. Any surplus PVCs will have to be manually deleted. The reason for skipping this functionality is that it can be tricky to select which PVCs to delete, and Union CSI does not actively need it in its current version.

6. Update the `status` field of the VolumeSplit and return non-nil error.

Listing 4.6: *The Reconcile method of the VolumeSplit Controller*

```
// The index key to configure the Manager's cache for looking up PVCs by owner name
var claimOwnerNameKey = "owner-name"

// VolumeSplitReconciler reconciles a VolumeSplit object
type VolumeSplitReconciler struct {
    client.Client
    Scheme *runtime.Scheme
    Log    logr.Logger
}

//+kubebuilder:rbac:groups=union.io,resources=volumesplits,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=union.io,resources=volumesplits/status,verbs=get;update;patch
//+kubebuilder:rbac:groups=union.io,resources=volumesplits/finalizers,verbs=update

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.

/*
VolumeSplit PVC reconciliation logic:

NOTE: this is a first iteration, a naive approach on how the controller manages its replicas.

  - On every reconciliation loop the reconciler fetches all the replicas (PVCs) that are owned
    by the reconciling object (VS) in the same namespace.
  - The reconciler does not inspect each replica to check if it matches the replica template
    or some kind of selector.
  - The reconciler does not fetch all the existing replicas in the same namespace as the owner
    and try to take ownership of orphaned replicas (replica that does not have an owner reference
    but match the selector) or release owned replicas that no longer match the selector.
    This is the kind of replica reconciliation logic found in many Kubernetes controllers that
    manage sets of replicas, like ReplicaSets.
*/
func (r *VolumeSplitReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    log := r.Log.WithValues("volumeSplit", req.NamespacedName.String())
    log.Info("Reconciling VolumeSplit")

    split := &v1alpha1.VolumeSplit{}
    if err := r.Get(ctx, req.NamespacedName, split); err != nil {
        if apierrors.IsNotFound(err) {
            log.Info("VolumeSplit not found, stopping reconciliation")
            return ctrl.Result{}, nil
        }
        log.Error(err, "Error getting VolumeSplit")
        return ctrl.Result{}, err
```

```
45      }
46
47      // Owner object is marked for deletion.
48      // The owner and owned objects are to be removed according to the specified cascading ↩
            deletion type.
49      // Stop the reconciliation loop here and let Kubernetes handle the GC.
50      if split.DeletionTimestamp != nil {
51          log.Info("VolumeSplit is marked for deletion, stopping reconciliation", "finalizers", ↩
                split.Finalizers)
52          return ctrl.Result{}, nil
53      }
54
55      claims := &corev1.PersistentVolumeClaimList{}
56      if err := r.List(ctx, claims, client.InNamespace(req.Namespace), client.MatchingFields{↩
            claimOwnerNameKey: req.Name}); err != nil {
57          log.Error(err, "Error listing replicas")
58          return ctrl.Result{}, err
59      }
60
61      if err := r.manageReplicas(ctx, log, split, getSliceOfClaimPointers(claims)); err != nil {
62          log.Error(err, "Error managing replicas")
63          return ctrl.Result{}, err
64      }
65
66      if err := r.updateVolumeSplitStatus(ctx, log, split, req); err != nil {
67          return ctrl.Result{}, err
68      }
69
70      return ctrl.Result{}, nil
71  }
72
73  func getSliceOfClaimPointers(claims *corev1.PersistentVolumeClaimList) []*corev1.↩
        PersistentVolumeClaim {
74      p := make([]*corev1.PersistentVolumeClaim, len(claims.Items))
75      for i := range claims.Items {
76          p[i] = &claims.Items[i]
77      }
78      return p
79  }
80
81  // manageReplicas reconciles replicas for split
82  // based on the desired number of replicas and the given number of claims
83  func (r *VolumeSplitReconciler) manageReplicas(ctx context.Context, log logr.Logger, split *↩
        v1alpha1.VolumeSplit, claims []*corev1.PersistentVolumeClaim) error {
84      desired, actual := int(*split.Spec.Replicas), len(claims)
85      n := desired - actual
86
87      if n > 0 {
88          log.Info("Scaling up replicas", "desired", desired, "actual", actual)
89          claim := getClaimFromTemplate(split)
90          if err := ctrl.SetControllerReference(split, claim, r.Scheme); err != nil {
91              return err
92          }
93          // Start creating replicas in batches that double in size with each successful batch
94          if err := slowStartBatch(n, 1, func() error {
95              // claim will be updated with the server response after a successful Create.
96              // Copy it before each time; otherwise, all calls will fail after the first one ↩
                    with a re-create error.
97              claim := claim.DeepCopy()
98              if err := r.Create(ctx, claim); err != nil {
99                  if apierrors.IsAlreadyExists(err) {
100                     log.Info("Replica already exists")
101                     return nil
102                 }
103                 log.Error(err, "Error creating replica")
104                 return err
105             }
106             // claim.Name has been generated here
107             claimKey := claim.Namespace + "/" + claim.Name
108             log.Info("Created replica", "persistentVolumeClaim", claimKey)
109             return nil
110         }); err != nil {
111             // At least one replica creation failed
112             return err
113         }
114     } else if n < 0 {
115         n *= -1
116         log.Info("Scaling down replicas", "desired", desired, "actual", actual)
117         log.Info("*** Scale down logic not yet implemented, aborting ***")
118     }
119
120     return nil
121 }
122
123 func getClaimFromTemplate(split *v1alpha1.VolumeSplit) *corev1.PersistentVolumeClaim {
124     claim := &corev1.PersistentVolumeClaim{}
125
126     claim.Namespace = split.Namespace
```

```
    claim.GenerateName = getClaimPrefix(split.Name)

    // Copy labels
    for k, v := range split.Spec.Template.Labels {
        claim.Labels[k] = v
    }
    // Copy annotations
    for k, v := range split.Spec.Template.Annotations {
        claim.Annotations[k] = v
    }
    // Copy finalizers
    claim.Finalizers = make([]string, len(split.Spec.Template.Finalizers))
    copy(claim.Finalizers, split.Spec.Template.Finalizers)

    // Copy spec from template
    claim.Spec = *split.Spec.Template.Spec.DeepCopy()

    return claim
}

// ValidatePersistentVolumeName checks that name is valid for PV/PVC objects.
// prefix indicates that name will be used for name generation, in which case
// trailing dashes are allowed.
var ValidatePersistentVolumeName = apimachineryvalidation.NameIsDNSSubdomain

func getClaimPrefix(name string) string {
    // Append trailing dash for prettier generated names if name is not too long
    prefix := name + "-"
    if len(ValidatePersistentVolumeName(prefix, true)) != 0 {
        prefix = name
    }
    return prefix
}

// slowStartBatch tries to call f a total of n times, starting slow
// to check for errors and fail early, then speeding up if calls succeed.
//
// It groups the calls into batches, starting with a batch of initialBatchSize.
// Within each batch, f will be called in a new goroutine.
//
// If a whole batch succeeds, the next batch will double in size.
// If there are any failures in a batch, all remaining batches are skipped
// after waiting for the current batch to complete and the first non-nil error
// of the batch is returned.
//
// NOTE: This function was lifted from github.com/kubernetes/kubernetes and slightly modified.
func slowStartBatch(n int, initialBatchSize int, f func() error) error {
    remaining := n
    min := func(x, y int) int {
        if x < y {
            return x
        }
        return y
    }
    for batchSize := min(remaining, initialBatchSize); batchSize > 0; batchSize = min(↩
         remaining, 2*initialBatchSize) {
        g := new(errgroup.Group)
        for i := 0; i < batchSize; i++ {
            g.Go(f)
        }
        if err := g.Wait(); err != nil {
            return err
        }
        remaining -= batchSize
    }
    return nil
}

// SetupWithManager sets up the controller with the Manager.
func (r *VolumeSplitReconciler) SetupWithManager(mgr ctrl.Manager) error {
    // Add index to group together PVCs with the same owner for efficient queries
    if err := mgr.GetFieldIndexer().IndexField(context.Background(), &corev1.↩
         PersistentVolumeClaim{}, claimOwnerNameKey, func(obj client.Object) []string {
        claim := obj.(*corev1.PersistentVolumeClaim)
        owner := metav1.GetControllerOfNoCopy(claim)
        if owner == nil {
            return nil
        }
        if owner.APIVersion != v1alpha1.GroupVersion.String() || owner.Kind != "VolumeSplit" {
            return nil
        }
        // Use the owner's Name as value
        return []string{owner.Name}
    }); err != nil {
        return err
    }

    return ctrl.NewControllerManagedBy(mgr).
```

```
213         For(&v1alpha1.VolumeSplit{}).
214         Owns(&corev1.PersistentVolumeClaim{}).
215         Complete(r)
216 }
```

4.2.3 Union CSI

Union CSI is the central part of the Union CSI software and this dissertation. It includes the CSI driver that implements the CSI gRPC services and integrates with Kubernetes, and the storage backend that is invoked by the CSI driver to carry out CRUD operations for Union CSI volumes using Kubernetes.

CSI driver

The part of Union CSI that implements the supported CSI RPCs, creates and listens on a CSI endpoint and responds to the RPCs made by the sidecar containers and Kubelet.

The design guidelines, supported capabilities, implemented RPCs and execution workflow of the CSI driver are thoroughly outlined and discussed in Chapter 3.

The command-line options of the Union CSI component are shown in Table 4.2 and mostly configure the CSI driver.

Option	Values	Default	Description
`--endpoint`		”unix:///tmp/csi.sock”	gRPC endpoint of CSI driver.
`--mode`	”controller”, ”node”, ”all”	”all”	Set CSI driver mode. ”controller” runs the Controller service, ”node” runs the Node service and ”all” runs both.
`--kubeconfig`			Absolute path to a kubeconfig file. Useful when running out-of-cluster.
`-v, --version`			Print version and exit.

Table 4.2: *”union-csi” command options*

The `--mode` option controls the set of RPC services the driver offers as a running instance. ”controller” configures the driver to register and serve the Controller and Identity services, ”node” to serve the Node and Identity services, and ”all” to serve all three

services. ”all” is the default, even though we only deploy the CSI driver using the ”controller” and ”node” modes.

Storage backend

Although referred to as a *backend*, it does not provide any HTTP endpoints or expose any API. It is an implementation of the Go `StorageBackend` interface (as designed) that is used by the CSI driver to manage VolumeSplit, PVC and Pod resources, abstracting any Kubernetes related details from it. The CSI driver utilizes the `StorageBackend` interface during the `CreateVolume`, `DeleteVolume`, `ControllerPublishVolume` and `ControllerUnpublishVolume` RPCs, forwarding the related request. We abuse the ”backend” term, as this interface functions as the ”storage backend” or ”storage service” that the CSI driver integrates with Kubernetes.

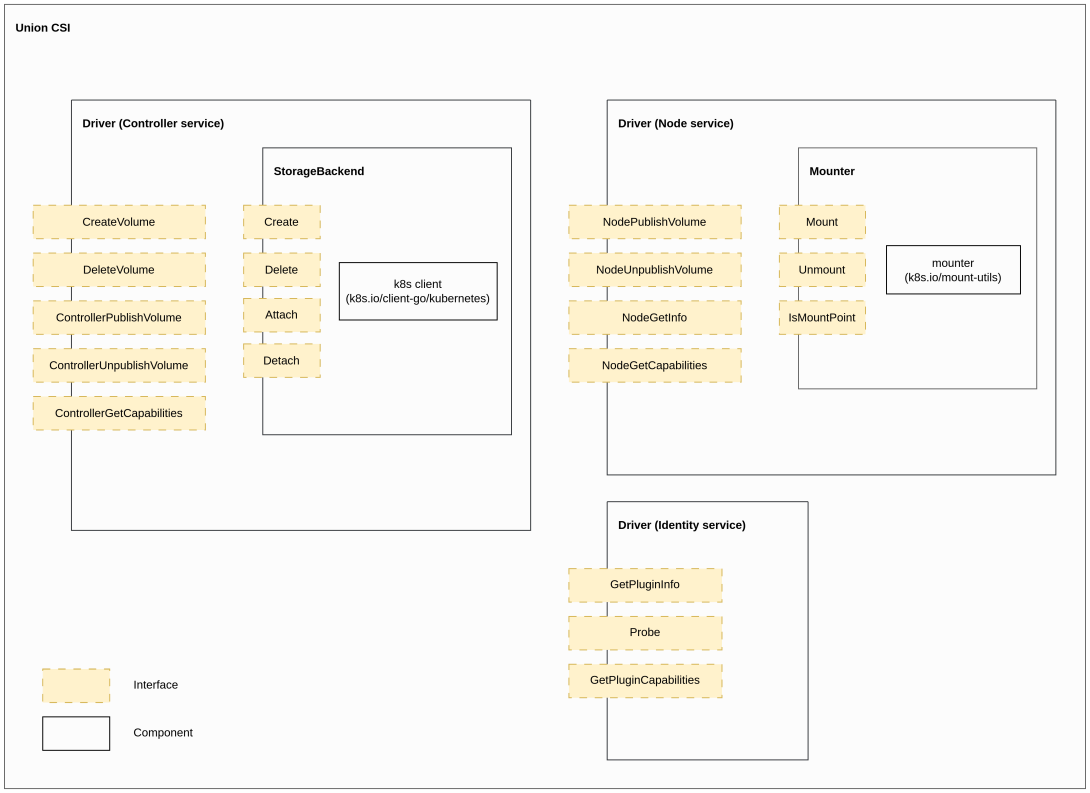


Figure 4.1: *The Union CSI internals*

The implementation follows exactly the design and algorithms outlined in Section 3.6.

For example, in the `CreateVolume` RPC, the CSI driver invokes the `Create` method of

the `StorageBackend`, using the `volume_id` along with other values obtained from the `CreateVolumeRequest`. In the `Create` method, the backend first checks if a volume with the given identifier already exists: It makes a `GET` request on the Kubernetes API server for a VolumeSplit object in the `union` namespace with the given identifier as a name. If the VolumeSplit exists, its `spec` payload is compared with the passed configuration for idempotency, returning an error if they are incompatible. If the VolumeSplit does not exist, then the backend performs a `CREATE` request on Kubernetes to create a new VolumeSplit as specified. In each (successful) case, the backend blocks for a short period to poll on the VolumeSplit instance for a ready state (status condition type of `Ready` or `Pending`). If the polling succeeds, the backend returns an internal representation of the VolumeSplit to the driver.

As part of the `CreateVolume`/`DeleteVolume` RPCs, the storage backend (`Create`/`Delete` methods) fetches, creates, watches and deletes VolumeSplit objects associated with the given volume identifier. It depends and waits on the VolumeSplit controller to create and delete the specified PersistentVolumeClaim objects.

As part of the `ControllerPublishVolume`/`ControllerUnpublish` RPCs, the backend (`Attach`/`Detach` methods) fetches the VolumeSplit objects associated with the given volume identifier, lists its owned PVCs, and fetches, creates, watches, and deletes the Pods (*Attach-Pods*) that use the PVCs of a VolumeSplit.

As part of the `NodePublishVolume`/`NodeUnpublishVolume` RPCs, the CSI driver employs the `Mounter` interface, as designed. The underlying implementation of the `Mounter` interface leverages the `mount-utils` [2] Kubernetes library for mounting, unmounting and probing filesystems.

Because the source code of the Union CSI RPCs, interfaces and functions expands across many lines of code, files and Go packages, we are not providing the code for the Union CSI component here.

[2] https://github.com/kubernetes/mount-utils

4.3 Deployment

In this section, we outline the deployment characteristics and resources, including container arguments and mounted volumes, for deploying the VolumeSplit Controller and Union CSI components in a Kubernetes environment. These deployments collectively form the Union CSI volume plugin.

4.3.1 VolumeSplit Controller

The VolumeSplit controller is packaged into a Docker image using the provided Makefile and Dockerfile. Kubebuilder automatically generates YAML manifests for essential deployment Kubernetes resources, such as ServiceAccount, ClusterRole, ClusterRoleBinding, and Deployment. Users have the flexibility to customize these manifests and promptly deploy the controller.

The provided Deployment manifest is modified to ensure the controller is deployed in the `union` namespace. In our deployment, a single replica Pod of the controller is installed, although it is generally advisable to use additional replicas for high availability. For replicated deployments, only one controller must be active at any given time, facilitated by the leader election functionality implemented by the `Manager` component of the controller. A `kustomization.yaml` file includes the VolumeSplit CRD and the controller Deployment so that both are conveniently applied and removed from a cluster.

4.3.2 Union CSI

The Union CSI component is containerized using its Dockerfile and deployed in Kubernetes within the `union` namespace using YAML manifests.

The Union CSI container can function as either a *Union CSI Controller* instance, a *Union CSI Node* instance, or an *all-in-one* instance, providing the Controller service, the Node service, or both, respectively. This behavior is controlled by the `--mode` option of the Union CSI component.

In our deployment, we provide YAML manifests that launch Union CSI as either *Controller* or *Node* instances. It is crucial to have at least one *Controller* instance running in

the entire cluster for volume provisioning/deletion and attachment/detachment. Additionally, at least one *Node* instance must be operational on each node where a provisioned volume is intended to be mounted/unmounted.

To achieve this, a single *Union CSI Controller* instance is provided through a Deployment resource, specifying one replica Pod.

Listing 4.7: *Deployment of Union CSI Controller*

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: union-csi-controller
  namespace: union
  labels:
    app.kubernetes.io/name: union-csi-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: union-csi-controller
  template:
    metadata:
      labels:
        app.kubernetes.io/name: union-csi-controller
    spec:
      nodeSelector:
        kubernetes.io/os: linux
      serviceAccount: union-service-account
      securityContext:
        runAsUser: 1000
        runAsGroup: 1000
        fsGroup: 1000
        runAsNonRoot: true
      containers:
      - name: union-csi
        image: docker.io/on2e/union-csi:dev
        imagePullPolicy: "Always"
        args:
        - --mode=controller
        - --endpoint=$(CSI_ENDPOINT)
        env:
        - name: CSI_ENDPOINT
          value: unix:///csi/csi.sock
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        volumeMounts:
        - name: socket-dir
          mountPath: /csi/
        securityContext:
          allowPrivilegeEscalation: false
      - name: external-provisioner
        image: docker.io/on2e/external-provisioner:v3.5.0
        imagePullPolicy: "IfNotPresent"
        args:
        - --csi-address=$(CSI_ENDPOINT)
        env:
        - name: CSI_ENDPOINT
          value: unix:///csi/csi.sock
        volumeMounts:
        - name: socket-dir
          mountPath: /csi/
        securityContext:
          allowPrivilegeEscalation: false
      - name: external-attacher
        image: docker.io/on2e/external-attacher:v4.3.0
        imagePullPolicy: "IfNotPresent"
        args:
        - --csi-address=$(CSI_ENDPOINT)
        env:
        - name: CSI_ENDPOINT
          value: unix:///csi/csi.sock
        volumeMounts:
        - name: socket-dir
          mountPath: /csi/
        securityContext:
          allowPrivilegeEscalation: false
      volumes:
      - name: socket-dir
        emptyDir:
```

The Deployment of Listing 4.7 includes:

- The following containers:
  - **`union-csi`**: The Union CSI master container serving the Controller service. The container is given the following arguments:
    * `--mode=controller`
    * `--endpoint=unix:///csi/csi.sock`
  - **`external-provisioner`**: The `external-provisioner` sidecar container. The container is given the following arguments:
    * `--csi-address=unix:///csi/csi.sock`
  - **`external-atttacher`**: The `external-atttacher` sidecar container. The container is given the following arguments:
    * `--csi-address=unix:///csi/csi.sock`
- The following volumes:
  - **`socket-dir`**: An `emptyDir` volume mounted on all three containers at `/csi/`. The `union-csi` container creates the UNIX domain socket under this directory and enables communication with the sidecar containers.

The *Union CSI Node* instances are provided through a DaemonSet resource, ensuring a Pod is running on every node.

Listing 4.8: *DaemonSet of Union CSI Node*

```yaml
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: union-csi-node
  namespace: union
  labels:
    app.kubernetes.io/name: union-csi-node
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: union-csi-node
  template:
    metadata:
      labels:
        app.kubernetes.io/name: union-csi-node
    spec:
      nodeSelector:
        kubernetes.io/os: linux
      serviceAccount: union-service-account
      securityContext:
        runAsUser: 0
        runAsGroup: 0
        fsGroup: 0
        runAsNonRoot: false
      containers:
      - name: union-csi
        image: docker.io/on2e/union-csi:dev
        imagePullPolicy: "Always"
        args:
        - --mode=node
        - --endpoint=$(CSI_ENDPOINT)
```

```
        env:
        - name: CSI_ENDPOINT
          value: unix:///csi/csi.sock
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        volumeMounts:
        - name: socket-dir
          mountPath: /csi/
        - name: kubelet-dir
          mountPath: /var/lib/kubelet/
          mountPropagation: Bidirectional
        - name: driver-dir
          mountPath: /var/lib/union-csi/
          mountPropagation: Bidirectional
        securityContext:
          privileged: true
      - name: node-driver-registrar
        image: docker.io/on2e/node-driver-registrar:v2.8.0
        imagePullPolicy: "IfNotPresent"
        args:
        - --csi-address=$(CSI_ENDPOINT)
        - --kubelet-registration-path=$(KUBELET_REGISTRATION_PATH)
        env:
        - name: CSI_ENDPOINT
          value: unix:///csi/csi.sock
        - name: KUBELET_REGISTRATION_PATH
          value: /var/lib/kubelet/plugins/union.csi.union.io/csi.sock
        volumeMounts:
        - name: registration-dir
          mountPath: /registration/
        - name: socket-dir
          mountPath: /csi/
        securityContext:
          allowPrivilegeEscalation: false
      volumes:
      - name: socket-dir
        hostPath:
          path: /var/lib/kubelet/plugins/union.csi.union.io/
          type: DirectoryOrCreate
      - name: registration-dir
        hostPath:
          path: /var/lib/kubelet/plugins_registry/
          type: Directory
      - name: driver-dir
        hostPath:
          path: /var/lib/union-csi/
          type: DirectoryOrCreate
      - name: kubelet-dir
        hostPath:
          path: /var/lib/kubelet/
          type: Directory
```

The DaemonSet of Listing 4.8 includes:

- The following containers:
  - **`union-csi`**: The Union CSI master container serving the Node service. The container runs with privileged security settings for mounting and unmounting volumes on the host filesystem. The container is given the following arguments:
    * `--mode=node`
    * `--endpoint=unix:///csi/csi.sock`
  - **`node-driver-registrar`**: The `node-driver-registrar` sidecar container. The container is given the following arguments:
    * `--csi-address=unix:///csi/csi.sock`
    * `--kubelet-registration-path=/var/lib/kubelet/plugins/union.csi.union.io/csi.sock`

- The following volumes:
  - **`socket-dir`**: A `hostPath` volume of `/var/lib/kubelet/plugins/union-.csi.union.io/` mounted on both the `union-csi` and `node-driver-registrar` containers at `/csi/`. `union-csi` creates the UNIX domain socket under this directory and the full host path is passed to the `node-driver-registrar` via the `--kubelet-registration-path` option.
  - **`registration-dir`**: A `hostPath` volume of `/var/lib/kubelet/plugins_-registry/` mounted on the `node-driver-registrar` container at `/registration/`. The `node-driver-registrar` creates a UNIX domain socket under this directory and informs Kubelet about `union-csi`'s `socket-dir` UNIX domain socket.
  - **`driver-dir`**: A `hostPath` volume of `/var/lib/union-csi/` mounted on the `union-csi` container at `/var/lib/union-csi/`. The MergerFS filesystems are published on the host under this directory by the Pods created by the *Union CSI Controller* using `Bidirectional` mount propagation. The `union-csi` container bind mounts these filesystems at the specified target paths during `NodePublishVolume`. This volume is also mounted with `Bidirectional` propagation so that `union-csi` receives the mounts from the MergerFS containers.
  - **`kubelet-dir`**: A `hostPath` volume of `/var/lib/kubelet/` mounted on the `union-csi` container at `/var/lib/kubelet/`. Kubelet generates unique target paths for `NodePublishVolume` under this directory and the `union-csi` container bind mounts to them the MergerFS filesystems under `/var/lib/union-csi/`. The volume is mounted with `Bidirectional` propagation so that `union-csi` receives updates from Kubelet and the MergerFS mounts are propagated to Kubelet.

Mount propagation: As described, CSI plugin Pods and Union CSI's MergerFS container Pods both utilize `Bidirectional` mount propagation to mount their volumes on the host filesystem. For mount propagation to work, the Container Runtime Interface (CRI) must allow shared mounts. More information on the mount propagation feature and how to check if it is enabled in a cluster can be found in the Kubernetes documen-

tation [3].

A *central* `kustomization.yaml` file includes both the VolumeSplit Controller and the Union CSI components's necessary deployment resources, enabling the entire application to be installed and uninstalled on a cluster with a single `kubectl` command.

Figure 4.2 shows the Union CSI component deployed as a *Union CSI Controller* and three *Union CSI Node* instances, accompanied by the required sidecar containers, along with the VolumeSplit Controller component, in a three-node cluster.

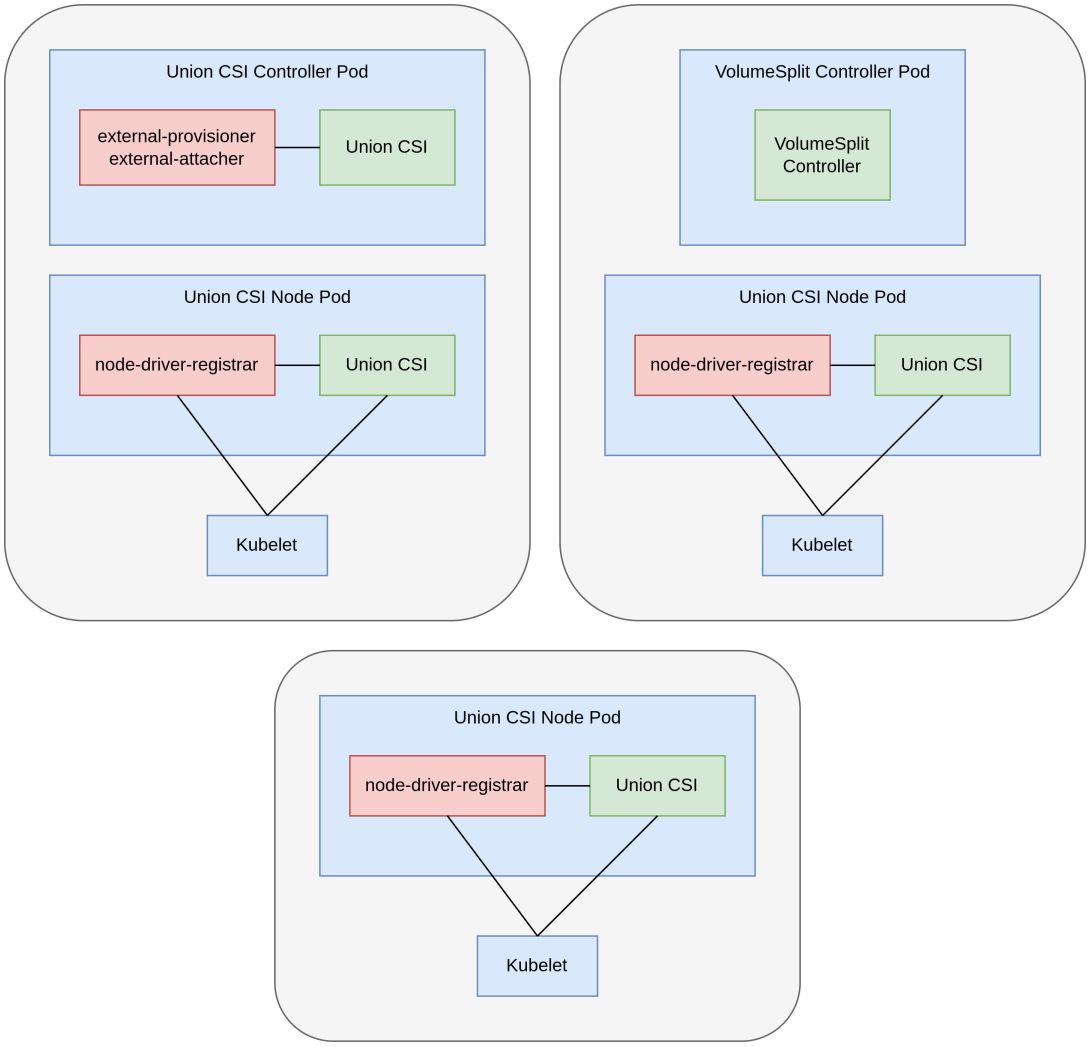


Figure 4.2: *The complete Union CSI deployment in a 3 node cluster*

[3]https://kubernetes.io/docs/concepts/storage/volumes/#mount-propagation

5 Evaluation

In this chapter, we demonstrate Union CSI and gain a preliminary understanding of its performance.

5.1 Overview

We demonstrate and evaluate the Union CSI plugin, along with established volume plugins for Kubernetes that leverage local storage on a node to provide persistent volumes for Pod containers.

To establish a baseline, we use the *Local Path Provisioner*, conducting tests directly on the host filesystem. Additionally, we employ *Longhorn*, an enterprise-grade, replicated block storage system, as our lower storage system. This allows us to showcase how Union CSI integrates with Longhorn, enabling us to harness a large volume whose total size would be unattainable otherwise.

In our testing, we explore these two plugins individually and with Union CSI layered on top. We assess metrics such as bandwidth and latency, aiming to understand the impact of the FUSE layer introduced by the MergerFS mount over the underlying volume pieces.

5.2 Description of the Volume Plugins

In this section, we present an overview of the Local Path Provisioner plugin and the Longhorn block storage system, discussing their design, architecture, and functionality.

5.2.1 Local Path Provisioner

Local Path Provisioner, developed by Rancher Labs, is a Kubernetes volume plugin designed to utilize the local storage in each node. It dynamically provisions either `hostPath` or `local` persistent volumes.

Unlike a CSI volume plugin, Local Path Provisioner does not integrate with Kubernetes through CSI and does not create `spec.csi` PersistentVolumes. Instead, it imports and utilizes the same external PersistentVolume controller dependency [1] as the `external-provisioner` sidecar. The Local Path Provisioner implements the `Provisioner` interface, passes the implementation to the external controller, and subsequently runs the controller. The controller actively monitors the Kubernetes API server for the creation and deletion of PVCs associated with a StorageClass that references the registered `Provisioner` and invokes the `Provision` and `Delete` methods, respectively.

During the `Provision` call, Local Path Provisioner launches a Pod on the specified node. This Pod mounts a parent directory from the node to the Pod container using a `hostPath` volume. The Pod container creates a directory under the container's mount path, a directory that is also visible on the host due to the `hostPath` volume. After completing its task, the container terminates, and the `Provision` method creates a PV to represent the newly created "volume." The PV is either `hostPath`- or `local`-backed and has the full node path in the `spec.hostPath.path` or `spec.local.path` field, respectively.

To contrast, in the `Provision` method of the `external-provisioner`, after the necessary preparations, the `external-provisioner` triggers the `CreateVolume` RPC on the given CSI endpoint and the CSI driver that is listening manages the provisioning process.

In the `Delete` call, Local Path Provisioner once again creates a Pod using `hostPath` to mount the same parent directory as in the `Provision` call for the respective volume.

[1] https://github.com/kubernetes-sigs/sig-storage-lib-external-provisioner

The container then proceeds to remove the child directory under the container's mount path, effectively cleaning up the host directory.

The Local Path Provisioner requires information about the node where the `hostPath` directory should be created to launch the Pod that sets up the directory. As a result, Local Path Provisioner operates exclusively with a StorageClass of delayed volume binding mode (`WaitForFirstConsumer`). Furthermore, since `hostPath` volumes are locally confined, the associated PV created by the `Provision` method has node affinity configured. This ensures that Pods utilizing the PV will only be scheduled on the node where the `hostPath` directory was initially created.

Local Path Provisioner ships with KIND (Kubernetes IN Docker) to enable dynamic provisioning of `hostPath` volumes in local computer clusters. Due to the many security risks and non-flexible nature of `hostPath` volumes, Local Path Provisioner is primarily intended for local clusters and testing scenarios.

5.2.2 Longhorn

Longhorn is a free, open-source, cloud-native, distributed block storage system designed for Kubernetes. It utilizes the local storage of Kubernetes nodes, eliminating the need for external storage arrays and cloud providers. Longhorn employs containers, microservices, and CRDs to implement distributed block storage.

Each volume in Longhorn is treated as a microservice, with a dedicated storage controller created for each block device volume. This storage controller, known as the *Longhorn Engine*, runs as a Linux process on the same node as the Pod using the Longhorn volume. The Longhorn Engine synchronously replicates the volume across multiple replicas stored on different nodes. Replicas, also considered microservices, are backed by thin-provisioned Linux sparse files.

Longhorn ensures high availability by having a separate storage controller for each volume and maintaining multiple replicas. If an issue arises with the Longhorn Engine process or a replica, it won't impact other volumes in the disk pool or replicas, allowing uninterrupted access for the Pod.

Figure 5.1 shows this design concept along with the data flow between the volumes, Longhorn Engines, replica instances, and the node disks.

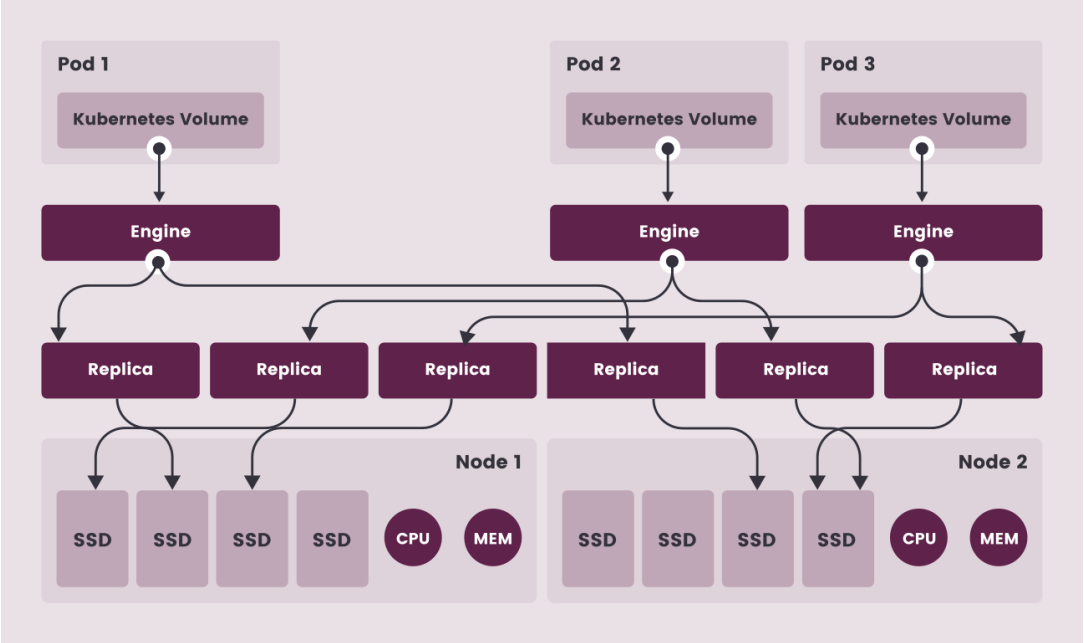


Figure 5.1: *How Longhorn works*

[Source: https://longhorn.io/docs/1.5.3/concepts/]

The *Longhorn Manager* orchestrates the Longhorn Engine and volumes. It operates as a per-node Pod responsible for creating, managing, and repairing Longhorn volumes, generating Longhorn Engine and replica instances, serving the Longhorn API, and communicating with the Kubernetes API server.

Longhorn is not a topology-aware system, meaning it does not require knowledge of a Pod's scheduled node to start provisioning a volume for it. As a result, the provided Longhorn StorageClass is defaulted to `Immediate` volume binding mode and the created volumes and PVs have no node constraints.

Additionally, for cross-node usability of volumes by Pods anywhere in the cluster, Longhorn utilizes iSCSI (Open-iSCSI). By default, Longhorn uses the Kubernetes cluster Container Network Interface (CNI) network.

Integration with Kubernetes is facilitated through Longhorn's CSI driver. This driver forwards create, delete, attach, and detach operations to the Longhorn Manager via its API. It also handles formatting, mounting, and unmounting of block device volumes on the nodes.

Additional key features of Longhorn include:

- Incremental snapshot of block storage.

- Store backup data in external storage, such as NFS or AWS S3.
- Restore volumes from backup.
- Upgrade Longhorn without disrupting persistent volumes.

Longhorn was originally developed by Rancher Labs and is now recognized as a Cloud Native Computing Foundation (CNCF) incubating project.

5.3 Demonstration of Union CSI with Longhorn

In this section, we show and use the Longhorn storage system (v1.5.1) and attempt to create a large enough volume that exceeds the available disk space on any individual node. Subsequently, we demonstrate how Union CSI ("dev") can be integrated with Longhorn to aggregate its volumes and achieve the desired capacity. Longhorn is a fitting match for Union CSI due to its ability to remotely access its volumes from any node in the cluster. This enables Union CSI to create PVCs for Longhorn volumes that span across different nodes and seamlessly attach them on the desired node.

Please note that the installation processes for Longhorn and Union CSI is not be provided in this section.

5.3.1 Environment

The Longhorn and Union CSI plugins are installed and tested in a two-node Kubernetes cluster managed by Azure's AKS (Azure Kubernetes Service). The nodes are general purpose virtual machines of the `Standard_DS2_v2` [2] tier, each equipped with 2 vCPU cores, 7GiB RAM, and a 128GiB local SSD. Each node serves dual roles as both a Kubernetes master (control-plane) and a worker node.

5.3.2 Longhorn StorageClass

The Longhorn installation includes a default StorageClass for creating Longhorn volumes using PVCs. The StorageClass has many configurable parameters, one of which is

[2] https://azureprice.net/vm/Standard_DS2_v2

the `numberOfReplicas` parameter that controls the number of replicas that Longhorn creates for a volume. The parameter accepts values between 1 and 20, with a default of 3.

For the purposes of this demonstration, it is important to set `numberOfReplicas` to one. This configuration ensures that only one primary replica volume is provisioned when creating a Longhorn volume, whether directly through `kubectl` or via the Union CSI plugin. This setting is essential in order to have Longhorn allocate the specified capacity and no more. Additionally, this reduced availability is desired for later tests, specifically when examining the network overhead for Union CSI when using Longhorn volumes remotely.

By setting `numberOfReplicas` to 1, the StorageClass used for Longhorn volumes in subsequent steps is illustrated in YAML in Listing 5.1. The `provisioner` field references the Longhorn CSI driver by the name `driver.longhorn.io`.

Listing 5.1: *Longhorn StorageClass for 1 primary replica volumes*

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: longhorn-1-replica
parameters:
  dataLocality: disabled
  fromBackup: ""
  fsType: ext4
  numberOfReplicas: "1"
  staleReplicaTimeout: "30"
provisioner: driver.longhorn.io
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

5.3.3 Creating a Large Longhorn Volume

We aim to test what will happen if we request from Longhorn a volume with a capacity that surpasses the available disk space of any node in the cluster. As a reminder, our Kubernetes cluster consists of two nodes, each equipped with a 128GiB SSD. Approximately 87GiB of storage is available on each node to be pooled by Longhorn.

To create a Longhorn volume, first we need to create a PVC using the StorageClass we described and installed in the previous section. Listing 5.2 demonstrates a PVC claiming a 120GiB Longhorn volume.

Listing 5.2: *PVC for a 120GiB Longhorn volume*

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: longhorn-pvc
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: longhorn-1-replica
  resources:
    requests:
      storage: 120Gi
```

After creating the PVC in the Kubernetes API server using `kubectl`, we inspect the object's status. Listing 5.3 showcases the relevant commands and the API responses in the command-line, showing the successful binding of the PVC to a PV.

Listing 5.3: *The Longhorn PVC is bound to a PV*

```
$ kubectl get pvc longhorn-pvc

NAME          STATUS  VOLUME                                     CAPACITY  ACCESS MODES  ↩
    STORAGECLASS        AGE
longhorn-pvc  Bound   pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74  120Gi     RWO           ↩
    longhorn-1-replica  3s

$ kubectl get pv

NAME                                       CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  ↩
    CLAIM                  STORAGECLASS        REASON  AGE
pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74  120Gi     RWO           Delete          Bound   ↩
    default/longhorn-pvc  longhorn-1-replica          4s
```

At first glance, it might seem that we have successfully created a 120GiB volume on a disk with 87GiB available space, since a PersistentVolume represents a piece of storage in Kubernetes. However, a closer examination reveals otherwise.

Observing the Longhorn Replica CR object corresponding to the created volume, we notice that the `NODE` and `DISK` column entries are missing a value, as shown in Listing 5.4. This indicates that Longhorn failed to schedule the replica on a node and allocate the required block device.

Listing 5.4: *The Longhorn Replica custom resource is missing NODE and DISK IDs*

```
$ kubectl get -n longhorn-system replica

NAME                                                STATE    NODE  DISK  INSTANCEMANAGER  ↩
    IMAGE  AGE
pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74-r-18051260  stopped ↩
                                                    9m4s
```

Additionally, inspecting the logging messages of the Longhorn Manager Pod which handled the provisioning request on behalf of the Longhorn CSI driver, reveals repeated reports stating insufficient available disk space for the requested replica, as shown in Listing 5.5

Listing 5.5: *The Longhorn Manager Pod logs "There's no available disk for replica ..."*

```
$ kubectl logs -n longhorn-system longhorn-manager-9d5m5

```

```
...
time="2023-11-26T17:47:27Z" level=error msg="There's no available disk for replica pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74-r-18051260, size 128849018880"
time="2023-11-26T17:47:27Z" level=warning msg="Failed to schedule replica" accessMode=rwo controller=longhorn-volume frontend=blockdev migratable=false node=aks-agentpool-30515490-vmss00000o owner=aks-agentpool-30515490-vmss00000o replica=pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74-r-18051260 state=detached volume=pvc-76b412fc-e2af-475f-85d6-3f9091cf5b74
...
```

The requested 120GiB Longhorn volume is not possible. If we create a Pod using the PVC, the Pod will also fail to get scheduled on a node and access the volume. In the following sections, we explore how Union CSI can help overcome this limitation, enabling the utilization of a 120GiB volume composed of two smaller, 60GiB Longhorn volumes.

5.3.4 Union CSI StorageClass

To configure Union CSI to use Longhorn as its underlying storage provisioner, we first need to create and install the appropriate StorageClass. The StorageClass in Listing 5.6 specifies the Union CSI driver name, `union.csi.union.io`, in the `provisioner` field and the Longhorn StorageClass name, `longhorn-1-replica`, in the `parameters` list under `lowerStorageClassName`. This way, Union CSI creates PVCs using the Longhorn StorageClass and has Longhorn create the lower volumes of the Union CSI volume.

Listing 5.6: *Union CSI StorageClass using the Longhorn StorageClass*

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: union-storage-longhorn
provisioner: union.csi.union.io
parameters:
  lowerStorageClassName: longhorn-1-replica
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

5.3.5 Creating a Union CSI Volume

After cleaning up the PVC created in Section 5.3.3, we proceed to create a new PVC requesting a 120GiB volume, this time using the Union CSI StorageClass. Upon creating the new PVC, the following sequence of events occurs:

1. The `CreateVolume` RPC of the Union CSI plugin is made.
2. During `CreateVolume`, Union CSI creates a VolumeSplit object specifying two replica PVCs of 60GiB each, utilizing the Longhorn StorageClass.

3. The VolumeSplit controller creates the desired number of lower PVCs using the PVC template specified on the VolumeSplit.
4. The `CreateVolume` RPC of Longhorn is made, with one call for each lower PVC.
5. During `CreateVolume`, Longhorn allocates a 60GiB volume on a node with available disk space.

Listing 5.7 illustrates a PVC utilizing the `union-storage-longhorn` StorageClass specified earlier and claiming a 120GiB Union CSI volume.

Listing 5.7: *PVC for a 120GiB Union CSI volume*

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: union-pvc-longhorn
spec:
  storageClassName: union-storage-longhorn
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 120Gi
```

After the PVC is successfully created, we check its status. Listing 5.8 verifies the following:

1. The upper PVC is bound to a 120GiB upper PV.
2. Union CSI created 2 lower PVCs (prefixed with `split-`), each of 60GiB, in the `union` namespace, using the Longhorn StorageClass.
3. Each lower PVC is bound to a 60GiB lower PV.

Listing 5.8: *The 120GiB upper PVC, the 120GiB upper PV, the two 60GiB lower PVCs and the two 60GiB lower PVs*

```
$ kubectl get pvc union-pvc-longhorn

NAME                STATUS  VOLUME                                     CAPACITY  ACCESS MODES  ↩
    STORAGECLASS           AGE
union-pvc-longhorn  Bound   pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0  120Gi     RWO           ↩
    union-storage-longhorn  12s

$ kubectl get -n union pvc

NAME                                          STATUS  VOLUME ↩
                                    CAPACITY  ACCESS MODES  STORAGECLASS        AGE
split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-k2p7n  Bound   pvc-75a2e35b-0488-41d0-ab2b-↩
    f85d0b35632c  60Gi      RWO           longhorn-1-replica  35s
split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-mst9w  Bound   pvc-bb282fc0-7d58-4ce1-872c-015↩
    eb05967c4  60Gi      RWO           longhorn-1-replica  35s

$ kubectl get pv

NAME                                       CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  ↩
    CLAIM                                                    STORAGECLASS            ↩
    REASON  AGE
pvc-75a2e35b-0488-41d0-ab2b-f85d0b35632c  60Gi      RWO           Delete          Bound   ↩
    union/split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-k2p7n  longhorn-1-replica ↩
                13s
```

```
16 pvc-bb282fc0-7d58-4ce1-872c-015eb05967c4  60Gi      RWO            Delete           Bound    ↩
       union/split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-mst9w  longhorn-1-replica ↩
                   13s
17 pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0  120Gi     RWO            Delete           Bound    ↩
       default/union-pvc-longhorn                                    union-storage-longhorn ↩
                16s
```

To confirm Longhorn's successful scheduling of replicas corresponding to the two Longhorn volumes requested by Union CSI, we fetch them from the Kubernetes API server. The output in Listing 5.9 indicates that the two replicas, one for each Longhorn volume, have been successfully scheduled on different nodes and assigned a block device.

Listing 5.9: *The Longhorn Replica custom resources corresponding to the two Longhorn volumes have NODE and DISK IDs*

```
1 $ kubectl get -n longhorn-system replica
2
3 NAME                                                STATE    NODE ↩
                                  DISK                                    INSTANCEMANAGER  ↩
      IMAGE  AGE
4 pvc-75a2e35b-0488-41d0-ab2b-f85d0b35632c-r-34eeca8c  stopped  aks-agentpool-30515490-↩
      vmss00000q  61233fcc-1261-41ff-9f0c-410c166ce168                        26m
5 pvc-bb282fc0-7d58-4ce1-872c-015eb05967c4-r-6208e7af  stopped  aks-agentpool-30515490-↩
      vmss00000p  d551e4b0-5ec9-422a-a5e3-04e66c948af4                        26m
```

Listing 5.10 displays the VolumeSplit CR created by Union CSI in the `union` namespace during the `CreateVolume` RPC. The two lower PVCs created by the VolumeSplit controller are also listed due to their linkage to the VolumeSplit instance through owner references.

Listing 5.10: *The VolumeSplit custom resource created by Union CSI and its two owned PVCs created by the VolumeSplit controller*

```
1 $ kubectl tree -n union volumesplits.union.io split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0
2
3 NAMESPACE  NAME
4 union      VolumeSplit/split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0
5 union      |-PersistentVolumeClaim/split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-k2p7n
6 union      +-PersistentVolumeClaim/split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-mst9w
```

Although the upper 120GiB PV is created in the API server, the underlying Union CSI volume does not exist yet, only its two 60GiB lower parts. To merge the two volumes and utilize the resulting volume on a node, we first need to create a consumer Pod.

5.3.6 Attaching & Mounting a Union CSI Volume

By creating a Pod that references the 120GiB PVC previously created, the following sequence of events occurs:

1. The Pod is scheduled on a node and the `ControllerPublishVolume` RPC of the Union CSI plugin is made.

2. During `ControllerPublishVolume`, Union CSI creates a new *Attach-Pod* in the `union` namespace. The Pod uses the two lower 60GiB PVCs claiming the Longhorn volumes, along with a `hostPath` volume. The *Attach-Pod* is assigned to the same node as the consumer Pod.
3. The `ControllerPublishVolume` RPC of Longhorn is made, with one call for each volume.
4. During `ControllerPublishVolume`, Longhorn attaches and mounts the two Longhorn volumes on the specified node—one volume locally and the other remotely.
5. Kubelet mounts the two Longhorn volumes and the `hostPath` directory on the *Attach-Pod* container and runs the container.
6. The MergerFS container of the *Attach-Pod* executes `mergerfs` (through the `gogo-mergerfs` program), combining the Longhorn filesystems and mounting the output on the `hostPath` container mount point. The MergerFS filesystem is propagated on the host due to the bidirectional mount propagation specified in the *Attach-Pod*.
7. Kubelet mounts the Union CSI volume (the MergerFS filesystem on the host) on the consumer Pod container and runs the container.

Listing 5.11 shows a Pod using the Union CSI volume through its PVC and specifying the `/data/` directory as the mount point inside an NGINX container.

Listing 5.11: *Pod using the Union CSI volume through its PVC*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: union-pod-longhorn
spec:
  containers:
  - name: app
    image: nginx:stable-alpine
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: vol
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: vol
    persistentVolumeClaim:
      claimName: union-pvc-longhorn
```

After the Pod is created, we can verify that both the user Pod and the *Attach-Pod* created by the Union CSI plugin are running on the API server, as shown in Listing 5.12.

Listing 5.12: *The consumer Pod and corresponding Attach-Pod are running*

```
$ kubectl get pod union-pod-longhorn

NAME                READY   STATUS    RESTARTS   AGE
union-pod-longhorn  1/1     Running   0          35s

$ kubectl get -n union pod pod-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0

NAME                                          READY   STATUS    RESTARTS   AGE
pod-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0  1/1     Running   0          43s
```

5.3.7 Inspecting & Utilizing a Union CSI Volume

Connecting to and executing commands within the user Pod container allows us to inspect its contents. Listing 5.13 displays the MergerFS filesystem mounted inside the NGINX container at `/data/`. Notice the unusually large inode numbers for `/data/` and its contents. This is because MergerFS defaults to setting inode numbers by hashing the corresponding relative paths and inodes of the underlying filesystems. Additionally, we use the `dd` utility to write two 10GiB files—`10GB.file` and `other10GB.file`—under `/data/`.

Listing 5.13: *The MergerFS filesystem within the consumer Pod container*

```
$ kubectl exec -it union-pod-longhorn -- /bin/sh
# ls -ild data

12101348768806803897 drwxr-xr-x    3 root     root          4096 Nov 28 17:33 data

# ls -ila /data

total 24
12101348768806803897 drwxr-xr-x    3 root     root          4096 Nov 28 17:33 .
             3873171 drwxr-xr-x    1 root     root          4096 Nov 28 17:33 ..
 4938303994640588972 drwx------    2 root     root         16384 Nov 28 17:33 lost+found

# dd if=/dev/zero of=/data/10GB.file bs=1M count=10240 status=progress
...
# dd if=/dev/zero of=/data/other10GB.file bs=1M count=10240 status=progress
...
# ls -ila /data

total 20971552
12101348768806803897 drwxr-xr-x 3 root root        4096 Nov 28 18:06 .
             3873171 drwxr-xr-x 1 root root        4096 Nov 28 17:33 ..
 9581384014260166741 -rw-r--r-- 1 root root 10737418240 Nov 28 18:06 10GB.file
 4938303994640588972 drwx------ 2 root root       16384 Nov 28 17:33 lost+found
17568198116042200688 -rw-r--r-- 1 root root 10737418240 Nov 28 18:08 other10GB.file
```

We can also attach to the MergerFS container within the *Attach-Pod*, where the MergerFS filesystem originates. In Listing 5.14, we observe the following:

1. The `/volume/` directory contains the `merged` and `branches` directories. The `/volume/merged/` directory has the same inode hash number and contains the same directories and files as the `/data/` directory in the NGINX container; they are the same filesystem.

2. The output of `mount` verifies that the MergerFS filesystem is mounted at `/volumes/merged/` with filesystem type of `fuse.mergerfs`. The filesystem name is composed by joining the branch names, separated by a colon, after trimming their common prefix.
3. Each Longhorn volume is mounted on its dedicated directory under `/volume/branches/` which is following the name scheme: `branch`, followed by an identification number, followed by a hyphen and the PVC name that corresponds to the volume.
4. The two 10GiB files created earlier in the MergerFS filesystem have been placed in the underlying filesystems: the first `10GB.file` has been created under the ”branch0” directory and the second `other10GB.file` under the ”branch1” directory. MergerFS's default policy is to choose the branch with the most free space when creating files while preserving relative existing paths (`epmf`[3] policy). So, when we created the second `other10GB.file` using `dd`, MergerFS chose the empty branch as its destination (since the file was not created under an existing path in the MergerFS mount).
5. The output of `ps` reveals the `mergerfs` daemon process that handles the FUSE messages (PID 16). PID 1 process is the `gogomergerfs` program that runs `mergerfs` and then blocks for UNIX signals (`SIGTERM` and `SIGINT`).

Listing 5.14: *The MergerFS fileysystem within the Attach-Pod container*

```
$ kubectl exec -it -n union pod-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0 -- /bin/sh
# ls -ila /volume

total 16
             1846804 drwxr-xr-x    4 root     root          4096 Nov 28 17:33 .
             1846803 drwxr-xr-x    3 root     root          4096 Nov 28 17:33 ..
             1846806 drwxr-xr-x    4 root     root          4096 Nov 28 17:33 branches
12101348768806803897 drwxr-xr-x    3 root     root          4096 Nov 28 18:06 merged

# mount | grep mergerfs
0-split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-k2p7n:1-split-pvc-ff5fca91-682b-48e9-959a-4↩
    a38ee5e2bf0-mst9w on /volume/merged type fuse.mergerfs (rw,nosuid,nodev,relatime,user_id↩
    =0,group_id=0,default_permissions,allow_other)

# ls -ila /volume/merged

total 20971552
12101348768806803897 drwxr-xr-x    3 root     root            4096 Nov 28 18:06 .
             1846804 drwxr-xr-x    4 root     root            4096 Nov 28 17:33 ..
 9581384014260166741 -rw-r--r--    1 root     root     10737418240 Nov 28 18:06 10GB.file
 4938303994640588972 drwx------    2 root     root           16384 Nov 28 17:33 lost+found
17568198116042200688 -rw-r--r--    1 root     root     10737418240 Nov 28 18:08 other10GB.file

# ls -ila /volume/branches/

total 16
1846806 drwxr-xr-x    4 root     root          4096 Nov 28 17:33 .
1846804 drwxr-xr-x    4 root     root          4096 Nov 28 17:33 ..
      2 drwxr-xr-x    3 root     root          4096 Nov 28 18:06 branch0-split-pvc-ff5fca91↩
          -682b-48e9-959a-4a38ee5e2bf0-k2p7n
```

[3] https://github.com/trapexit/mergerfs#policies

```
      2 drwxr-xr-x    3 root     root          4096 Nov 28 18:05 branch1-split-pvc-ff5fca91↩
          -682b-48e9-959a-4a38ee5e2bf0-mst9w

# ls -ila /volume/branches/branch0-split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-k2p7n/

total 10485788
      2 drwxr-xr-x    3 root     root            4096 Nov 28 18:06 .
1846806 drwxr-xr-x    4 root     root            4096 Nov 28 17:33 ..
     11 drwx------    2 root     root           16384 Nov 28 17:33 lost+found
     12 -rw-r--r--    1 root     root     10737418240 Nov 28 18:08 other10GB.file

# ls -ila /volume/branches/branch1-split-pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0-mst9w/

total 10485788
      2 drwxr-xr-x    3 root     root            4096 Nov 28 18:05 .
1846806 drwxr-xr-x    4 root     root            4096 Nov 28 17:33 ..
     12 -rw-r--r--    1 root     root     10737418240 Nov 28 18:06 10GB.file
     11 drwx------    2 root     root           16384 Nov 28 17:33 lost+found

# ps | grep mergerfs

    1 root      0:00 gogomergerfs mergerfs --branches=/volume/branches/* --target=/volume/↩
        merged --block
   16 root      0:23 mergerfs /volume/branches/* /volume/merged
   34 root      0:00 grep mergerfs
```

To conclude, we connect to the host of the node where the Longhorn and Union CSI volumes are mounted and the consumer Pod and *Attach-Pod* are running. In Listing 5.15, we observe the following:

1. The MergerFS filesystem is mounted on the host at `/var/lib/union-csi/volumes/<volume_id>/merged/`.
2. The two Longhorn block devices are located under `/dev/longhorn/`.
3. The `gogomergerfs` and `mergerfs` processes are also running on the host, each with different process IDs than the privileged MergerFS container.

Listing 5.15: *The MergerFS filesystem and the Longhorn block devices on the host node*

```
# ls -ila /var/lib/union-csi/volumes/pvc-ff5fca91-682b-48e9-959a-4a38ee5e2bf0/merged

total 20971552
12101348768806803897 drwxr-xr-x 3 root root        4096 Nov 28 18:06 .
            1840587 drwxr-xr-x 3 root root        4096 Nov 28 17:33 ..
 9581384014260166741 -rw-r--r-- 1 root root 10737418240 Nov 28 18:06 10GB.file
 4938303994640588972 drwx------ 2 root root       16384 Nov 28 17:33 lost+found
17568198116042200688 -rw-r--r-- 1 root root 10737418240 Nov 28 18:08 other10GB.file

# ls -l /dev/longhorn

total 0
brw-rw---- 1 root root 8, 32 Nov 28 17:33 pvc-75a2e35b-0488-41d0-ab2b-f85d0b35632c
brw-rw---- 1 root root 8, 48 Nov 28 17:33 pvc-bb282fc0-7d58-4ce1-872c-015eb05967c4

# ps aux | grep mergerfs

root       39863  0.0  0.1 712488  8708 ?        Ssl  17:33   0:00 gogomergerfs mergerfs --↩
    branches=/volume/branches/* --target=/volume/merged --block
root       39885  0.2  0.0   6092  3564 ?        S<sl 17:33   0:23 mergerfs /volume/branches/*↩
     /volume/merged
root      373769  0.0  0.0   3468  1656 ?        S+   20:30   0:00 grep mergerfs
```

5.3.8 Summary

By integrating Union CSI with Longhorn, we have successfully created, attached, mounted, and utilized a 120GiB volume, composed of two 60GiB Longhorn volumes allocated on different nodes.

Initially, creating a 120GiB Longhorn block device volume was not feasible on a cluster with 128GiB local SSDs, as it exceeded the available disk space. By incorporating the Union CSI volume plugin, the 120GiB volume request was split in half through storage claims (PVCs) made on the Longhorn system, allowing Longhorn to successfully provision the two smaller 60GiB storage pieces. Union CSI then merged the two lower halves using the MergerFS tool and mounted the union filesystem on the specified node. Any underlying filesystems on nodes remote to the target node were accessed by Longhorn through its iSCSI feature. While the Union CSI volume is mounted on a consumer Pod, the `mergerfs` root process running on the host node (executed in the privileged *Attach-Pod* created by the Union CSI driver) handles the IO requests of the Pod and forwards them to the underlying Longhorn filesystems.

In the next section, we will assess and compare the IO performance of the different storage systems.

5.4 Performance Measurements

In this section, we present relative performance results, in terms of measured bandwidth and latency, for Local Path Provisioner, Longhorn and Union CSI volumes, across various test cases. Through these tests, we aim to gain an insight of the impact of the MergerFS and network layers on the underlying filesystems.

5.4.1 Environment

The environment used is the same one that we used and described in Section 5.3.

5.4.2 Test Cases

The test cases involve measuring the IO performance of the volumes provided by the Local Path Provisioner, Longhorn and Union CSI plugins. Each plugin is tested individually or in combination with Union CSI, where the tested volume consists of the `mergerfs` output of two equally sized volumes provided by the underlying plugin. The options of the `mergerfs` program were left to their defaults. For Longhorn, tests are conducted both with and without interpolating storage networking by assigning the benchmarking Pod on the opposite or the same node where the Longhorn volume was allocated.

In each test case, we launch a Pod running a container with our filesystem benchmarking tool that performs the specified IO workload on the tested volume. The ”labels” and descriptions of each test case are provided in the following paragraphs.

Local: To establish a standard for comparison, it is crucial to measure read and write performance on a bare directory on the filesystem of the node. The Local Path Provisioner creates a directory on the host node that is mounted on the container containing our benchmarking tool, enabling us to measure IO metrics in this scenario. It is expected that this scenario will yield the best IO metrics.

Longhorn: We configure Longhorn to create only one primary replica when creating a PVC. The volume is allocated as a sparse file on one of the two nodes. In this scenario, we measure the performance of a Longhorn volume provisioned on the same node as the Pod is scheduled on. This will showcase the Longhorn block-level engine at its finest.

Longhorn + iSCSI: Same scenario as before, but now we constrain our benchmarking Pod to run on the opposite node (by adding the appropriate `nodeSelector`) and expose the Longhorn block device on it through iSCSI. By doing so, we can quantify the added network delay of SCSI over Ethernet in our 2 node cluster.

Union + Local: Using the Local Path Provisioner as the underlying provisioner of the Union CSI plugin has no practical benefits. The two lower `hostPath` volumes will necessarily be provisioned on the same node, so there is no way to take advantage of disk space from different nodes with Union CSI. However, by using plain host directories for the MergerFS branches, we measure the impact of the FUSE userspace filesystem layer directly on the host filesystem.

Union + Longhorn: In this scenario, we once again use Longhorn as the lower CSI volume plugin for Union CSI, just as we did in Section 5.3. However, this time, we want the Longhorn volumes to be provisioned on the same node. For this reason, we create a small enough Union CSI volume that fits the node's disk, such as 30GiB split into two 15GiB lower Longhorn volumes. We then assign the benchmarking Pod to the same node as the lower volumes to assess the performance of the union mount of two local Longhorn volumes.

Union + Longhorn + iSCSI: Same scenario as before, but now the Pod is assigned to the opposite node than the one the two Longhorn volumes are scheduled on. This means that any requested IO has to go through the FUSE level, the iSCSI implementation and network level, and the Longhorn block device level. This represents a more real-world scenario for Union CSI where at least one of each branches will be remotely attached.

5.4.3 Test Tool & Configuration

For our testing, the fio tool was used. fio is a benchmarking and workload simulation tool that spawns a number of threads to perform configurable IO actions, typically provided through a job file.

The testing was configured to only perform random reads and writes with an IO block size of 4KiB on a 10GiB file. Furthermore, in order to ensure that the filesystems and devices are actually being tested and the OS stays as uninvolved as possible, the following settings were configured:

- 1 worker thread (`numjobs=1` parameter of fio).
- Queue depth of 1 (`iodepth=1`).
- Disabled page cache (`direct=1`).
- Synchronized reads and writes (`ioengine=sync`).

The above settings will intentionally limit the IO performance for all contestants, but it is necessary in order to focus on each component as narrowly as possible. For this reason, the absolute individual numbers do not matter, we will concentrate on their relative comparison.

5.4.4 Results

Measured bandwidth and latency results are normalized to the *Local* test case—all values are shown as a multiple of *Local*'s value.

Figure 5.2 illustrates the random read bandwidth performance for each test case, utilizing a 4KiB block size. As anticipated, *Local* exhibits the highest performance among all contestants. Following closely is *Longhorn*, with a 17% decrease in random read performance, further reduced to 47% in *Longhorn + iSCSI* due to the additional SCSI and network overhead. In the *Union* tests, all three cases provided nearly identical results, hovering around 30% of the performance of *Local*. Union CSI's random read results, although the lowest among all, did not experience further degradation, whether using simple host directories or Longhorn's block-level storage with remote access.

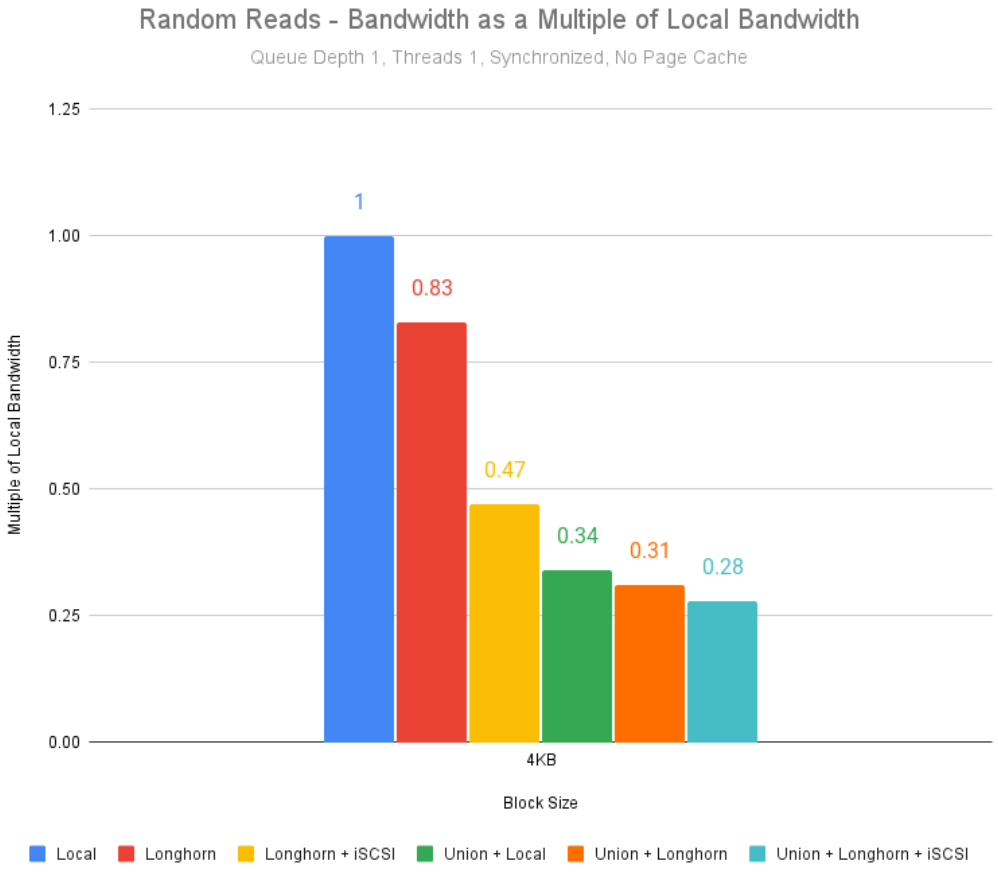


Figure 5.2: *Random read bandwidth performance for 4KiB block size*

Figure 5.3 illustrates the random write bandwidth performance for each test case, uti-

lizing a 4KiB block size. Both *Longhorn* and *Union + Local* almost reach the threshold of *Local*, while the two cases with the network-attached volumes, *Longhorn + iSCSI* and *Union + iSCSI*, have the lowest scores at 74% and 64%, respectively. In the *Union* cases, replacing Local Path Provisioner with Longhorn as the lower provisioner (*Union + Longhorn*) results in a performance decrease of around 19%, with a further 17% decrease compared to *Union + Longhorn*, when data transferring takes place (*Union + Longhorn + iSCSI*).

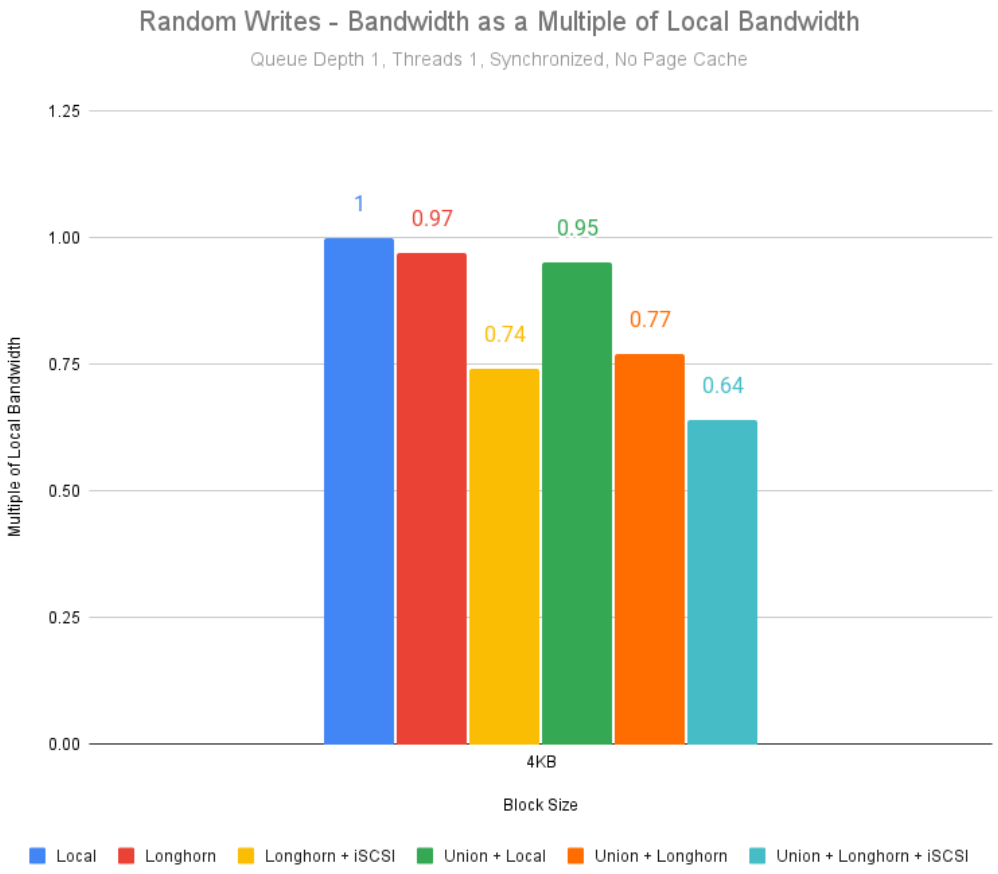


Figure 5.3: *Random write bandwidth performance for 4KiB block size*

Figure 5.4 illustrates the random read latency performance for each test case, utilizing a 4KiB block size. We observe that, in our environment and configuration, randomly reading from a local Longhorn volume is 1,65 times slower than *Local*, while from a remote Longhorn volume is up to 3 times slower. Similar to the random read bandwidth results, the *Union* tests all exhibit a similar level of performance, around 5 times slower than *Local*.

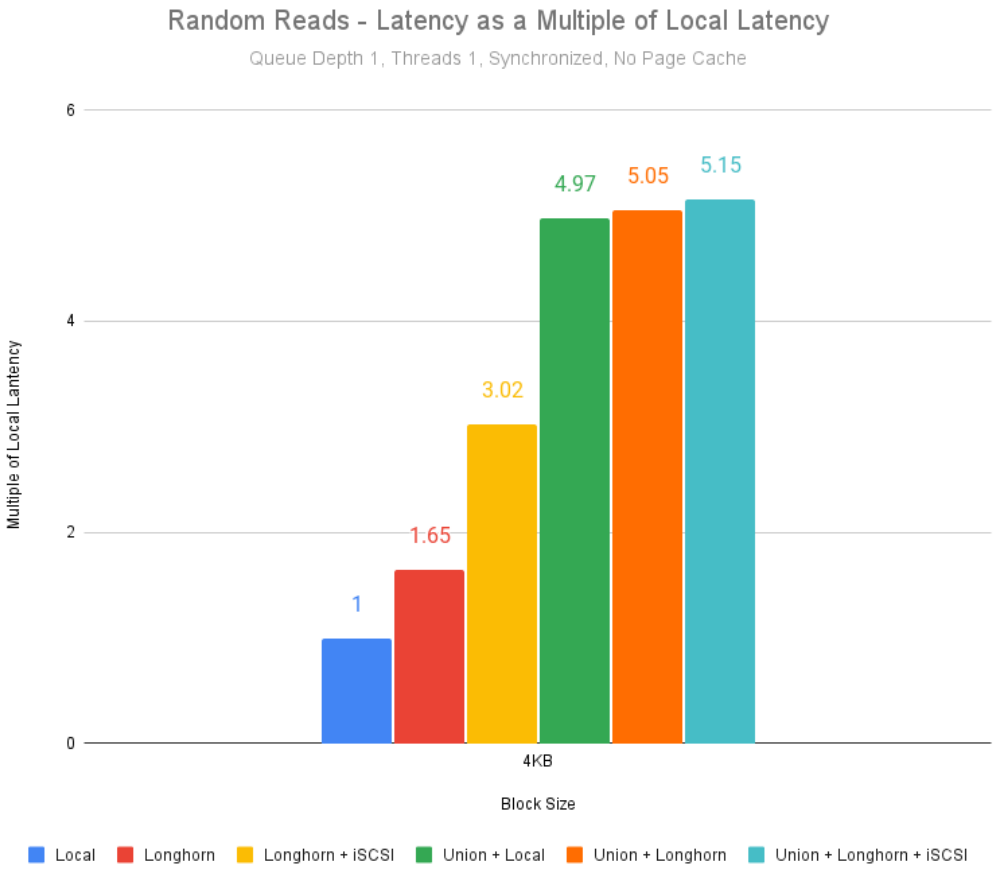


Figure 5.4: *Random read latency performance for 4KiB block size*

Figure 5.5 illustrates the random write latency performance for each test case, utilizing a 4KiB block size. As with random write bandwidth scores, *Longhorn* and *Union + Local* achieve practically the same performance as *Local*. Using local Longhorn volumes with Union CSI is 1.3 times slower than *Local*, while the network cases are the slowest, with *Union + Longhorn + iSCSI* being 1.56 times slower than *Local*.

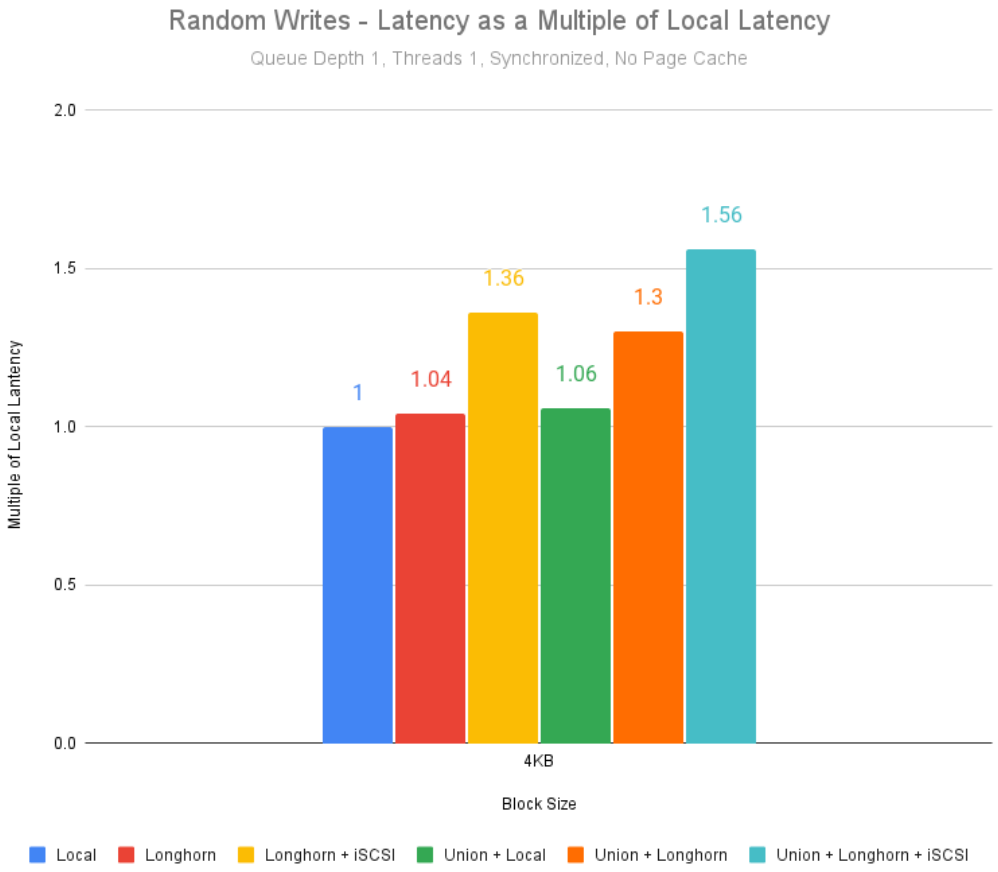


Figure 5.5: *Random write latency performance for 4KiB block size*

5.4.5 Summary

The relatively slower performance observed for Union CSI volumes was expected, due to the FUSE layer over the underlying volumes, particularly in the case where data transferring occurred through iSCSI for the Longhorn volumes.

MergerFS is a FUSE userspace program acting as a proxy for its underlying filesystems. IO requests on the FUSE filesystem incur overhead in transitioning between userspace and kernelspace. The more transitions needed, the worse the performance. One approach to reduce the number of transitions is to increase the IO block size. In our tests, we set a small block size of 4KiB to focus on random IO. Using a larger block size along with enabling OS page caching could significantly enhance Union CSI performance, even to a greater degree than the other plugins.

Moreover, the MergerFS author highlights various reasons for reduced performance in MergerFS mounts and provides insights into benchmarking techniques and parameter adjustments [4]. MergerFS offers numerous configurable options, many of which impact performance. A deeper understanding of these options and their fine-tuning could lead to better results than those presented in our tests.

In summary, even the most sophisticated userspace application has limitations in maintaining increased performance and minimizing user-kernel context switching. The results of our tests underscore the need for a kernel-based union filesystem solution for enterprise storage workloads.

[4] https://github.com/trapexit/mergerfs#performance

6 Conclusion

We are approaching the final pages of this thesis. In this chapter, we summarize the motivations and key aspects behind the proposed mechanism presented in this thesis. Subsequently, we outline our future work list, understanding our current position and establishing new goals.

6.1 Concluding Remarks

The primary objective of this thesis and its associated project was to explore and implement an innovative storage solution. This solution aims to support multi-TB filesystems by combining smaller storage pieces, overcoming the capacity and scalability limitations of existing local storage Kubernetesclusters.

Initially, we introduced Union CSI as a ”meta-plugin”, functioning in collaboration with other storage plugins to pool their volumes. A comprehensive analysis was provided on the architecture, features, and design choices governing Union CSI that make it a Kubernetes native, CSI-compliant solution that provides storage as union mounts. We proceeded to outline the implementation of the various Union CSI components, inspected some of its source code and delved into some of the deployment details of CSI drivers. To test Union CSI and evaluate its performance, we demonstrated the creation and utilization of an ”oversized” Union CSI volume using Longhorn. A comparison of their relative IO throughput and latency was presented. While benchmark results did not favor Union CSI—and real-world IO workloads will most probably not yield such limited performance—we can now confidently advocate for the inclusion of MergerFS

in the standard Linux kernel.

The project's development was an immensely enlightening and transformative experience for those involved. Through this paper, we aspired to convey some of the enthusiasm and knowledge gained.

6.2 Future Work

Concluding this initial, primarily experimental, and education-focused version of the Union CSI software, along with assessing the knowledge and expertise acquired, we have gained a clear perspective on potential improvements. Aside from addressing specific cases (e.g., boosting the uptime of MergerFS mounts by employing a Deployment or StatefulSet instead of a bare Pod, enriching the VolumeSplit API to convey more detailed information about the replica PVC statuses) and spending more time on testing, several new design and implementation directions emerge, including:

- Make Union CSI considerate of the available disk space in each node, enabling intelligent splitting and scheduling decisions.
- Evolve Union CSI into an end-to-end, independent storage system. Currently, Union CSI relies on the underlying storage system for its data path and storage networking features, with no direct means to observe the results other than implicitly through the Kubernetes API. This design choice was primarily driven by the need for the timely completion of this diploma thesis and, admittedly, is peculiar, not robust, and susceptible to multiple points of failure. The concepts and strengths of Union CSI and union filesystems should be integral to a larger, feature-rich, self-contained storage management solution.
- Subject MergerFS to stress tests by combining and managing hundreds of lower volumes. Ultimately, create a proof-of-concept implementation of MergerFS in the Linux kernel, compare its performance with the userspace implementation, and use the results to advocate for its inclusion in the mainline Linux kernel.

Bibliography

[1] Saad Ali. ”CSI Volume Plugins in Kubernetes Design Doc”. *GitHub*. https://github.com/kubernetes/design-proposals-archive/blob/main/storage/container-storage-interface.md. [Accessed on Dec. 27, 2023].

[2] Valerie Aurora. ”Unioning file systems: Architecture, features, and design choices”. *LWN.net*. https://lwn.net/Articles/324291/, Mar. 2009. [Accessed on Dec. 28, 2023].

[3] Container Storage Interface (CSI) Specification Authors. ”CSI specification Go language bindings [Go source code, v1.8.0]”. *GitHub*. https://github.com/container-storage-interface/spec/blob/v1.8.0/lib/go/csi/csi.pb.go. [Accessed on Dec. 27, 2023].

[4] Longhorn Authors. ”Architecture and Concepts”. *Longhorn Documentation*. https://longhorn.io/docs/1.5.3/concepts/. [Accessed on Dec. 10, 2023].

[5] Longhorn Authors. ”Longhorn”. *GitHub*. https://github.com/longhorn/longhorn. [Accessed on Dec. 10, 2023].

[6] Longhorn Authors. ”What is Longhorn?”. *Longhorn Documentation*. https://longhorn.io/docs/1.5.3/what-is-longhorn/. [Accessed on Dec. 10, 2023].

[7] The Kubernetes Authors. ”API Conventions”. *GitHub*. https://github.com/kubernetes/community/blob/

fb55d44be24fa626d38c9116e966c0237ecd58ab/contributors/devel/sig-architecture/api-conventions.md. [Accessed on Dec. 27, 2023].

[8] The Kubernetes Authors. "API Overview". *Kubernetes Documentation*. https://kubernetes.io/docs/reference/using-api/. [Accessed on Dec. 27, 2023].

[9] The Kubernetes Authors. "Cloud Controller Manager". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/architecture/cloud-controller/. [Accessed on Dec. 27, 2023].

[10] The Kubernetes Authors. "Command line tool (kubectl)". *Kubernetes Documentation*. https://kubernetes.io/docs/reference/kubectl/. [Accessed on Dec. 27, 2023].

[11] The Kubernetes Authors. "Controllers". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/architecture/controller/. [Accessed on Dec. 27, 2023].

[12] The Kubernetes Authors. "CSI attacher". *GitHub*. https://github.com/kubernetes-csi/external-attacher. [Accessed on Dec. 28, 2023].

[13] The Kubernetes Authors. "CSI provisioner". *GitHub*. https://github.com/kubernetes-csi/external-provisioner. [Accessed on Dec. 28, 2023].

[14] The Kubernetes Authors. "CSINode". *Kubernetes Documentation*. https://kubernetes.io/docs/reference/kubernetes-api/config-and-storage-resources/csi-node-v1/. [Accessed on Dec. 28, 2023].

[15] The Kubernetes Authors. "Custom Resources". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. [Accessed on Dec. 27, 2023].

[16] The Kubernetes Authors. "Declarative Management of Kubernetes Objects Using Kustomize". *Kubernetes Documentation*. https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/. [Accessed on Dec. 5, 2023].

[17] The Kubernetes Authors. "Device plugin registration". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/extend-kubernetes/

compute-storage-net/device-plugins/#device-plugin-registration. [Accessed on Dec. 28, 2023].

[18] The Kubernetes Authors. "Dynamic Volume Provisioning". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/. [Accessed on Dec. 27, 2023].

[19] The Kubernetes Authors. "Extend the Kubernetes API with CustomResourceDefinitions". *Kubernetes Documentation*. https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/. [Accessed on Dec. 27, 2023].

[20] The Kubernetes Authors. "KIND". *GitHub*. https://github.com/kubernetes-sigs/kind. [Accessed on Dec. 17, 2023].

[21] The Kubernetes Authors. "Kubebuilder". *GitHub*. https://github.com/kubernetes-sigs/kubebuilder. [Accessed on Dec. 5, 2023].

[22] The Kubernetes Authors. "Kubernetes API Concepts". *Kubernetes Documentation*. https://kubernetes.io/docs/reference/using-api/api-concepts/. [Accessed on Dec. 27, 2023].

[23] The Kubernetes Authors. "Kubernetes Components". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/overview/components/. [Accessed on Dec. 27, 2023].

[24] The Kubernetes Authors. "Kubernetes CSI Developer Documentation". https://kubernetes-csi.github.io/docs/introduction.html. [Accessed on Dec. 28, 2023].

[25] The Kubernetes Authors. "Kubernetes-to-CSI access modes [Go source code]". *GitHub*. https://github.com/kubernetes-csi/csi-lib-utils/blob/v0.16.0/accessmodes/access_modes.go. [Accessed on Dec. 27, 2023].

[26] The Kubernetes Authors. "Kustomize". *GitHub*. https://github.com/kubernetes-sigs/kustomize. [Accessed on Dec. 5, 2023].

[27] The Kubernetes Authors. "Mount propagation". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/volumes/#mount-propagation. [Accessed on Dec. 26, 2023].

[28] The Kubernetes Authors. "mount-utils". *GitHub*. https://github.com/kubernetes/mount-utils. [Accessed on Dec. 5, 2023].

[29] The Kubernetes Authors. "Namespaces". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. [Accessed on Dec. 27, 2023].

[30] The Kubernetes Authors. "Node Driver Registrar". *GitHub*. https://github.com/kubernetes-csi/node-driver-registrar. [Accessed on Dec. 28, 2023].

[31] The Kubernetes Authors. "Objects In Kubernetes". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/overview/working-with-objects/. [Accessed on Dec. 27, 2023].

[32] The Kubernetes Authors. "Operator pattern". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. [Accessed on Dec. 27, 2023].

[33] The Kubernetes Authors. "Overview". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/overview/. [Accessed on Dec. 27, 2023].

[34] The Kubernetes Authors. "Persistent Volumes". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. [Accessed on Dec. 27, 2023].

[35] The Kubernetes Authors. "Pod Licecycle". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. [Accessed on Dec. 5, 2023].

[36] The Kubernetes Authors. "Pods". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed on Dec. 27, 2023].

[37] The Kubernetes Authors. "Storage Classes". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/storage-classes/. [Accessed on Dec. 27, 2023].

[38] The Kubernetes Authors. "The Kubebuilder Book". https://book.kubebuilder.io/. [Accessed on Dec. 5, 2023].

[39] The Kubernetes Authors. "The Kubernetes API". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. [Accessed on Dec. 27, 2023].

[40] The Kubernetes Authors. "Volumes". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/volumes/. [Accessed on Dec. 27, 2023].

[41] The Kubernetes Authors. "Workloads". *Kubernetes Documentation*. https://kubernetes.io/docs/concepts/workloads/. [Accessed on Dec. 27, 2023].

[42] The Kubernetes Authors. "Writting Controllers". *GitHub*. https://github.com/kubernetes/community/blob/6690abcd6b833f46550f5eaba2ec17a9e39b35c4/contributors/devel/sig-api-machinery/controllers.md. [Accessed on Dec. 27, 2023].

[43] The Kubernetes Authors. "Container Storage Interface (CSI) for Kubernetes GA". *Kubernetes Documentation*. https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/, Jan. 2019. [Accessed on Dec. 28, 2023].

[44] IBM Corporation. "iSCSI overview". *IBM Documentation*. https://www.ibm.com/docs/en/flashsystem-7x00/8.4.x?topic=pc-iscsi-overview. [Accessed on Dec. 10, 2023].

[45] etcd Authors. "Data model". *etcd Documentation*. https://etcd.io/docs/v3.5/learning/data_model/. [Accessed on Dec. 27, 2023].

[46] Gluster. "Architecture". *Gluster Docs*. https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/. [Accessed on Jan. 08, 2024].

[47] Gluster. "DEPRECATED: Gluster Container Storage Interface (CSI) driver". *GitHub*. https://github.com/gluster/gluster-csi-driver. [Accessed on Jan. 08, 2023].

[48] gRPC Authors. "Introduction to gRPC". *gRPC Documentation*. https://grpc.io/docs/what-is-grpc/introduction/. [Accessed on Dec. 28, 2023].

[49] Red Hat. "Red Hat Gluster Storage Life Cycle". *Red Hat*. https://access.redhat.com/support/policy/updates/rhs. [Accessed on Jan. 08, 2024].

[50] The kernel development community. "FUSE". *The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/filesystems/fuse.html. [Accessed on Dec. 28, 2023].

[51] Rancher Labs. "Local Path Provisioner". *GitHub*. https://github.com/rancher/local-path-provisioner/. [Accessed on Dec. 10, 2023].

[52] libfuse Authors. "libfuse". *GitHub*. https://github.com/libfuse/libfuse, 2001. [Accessed on Dec. 28, 2023].

[53] Microsoft. "Dv2 and DSv2-series". *Azure Virtual Machines Documentation*. https://learn.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series. [Accessed on Dec. 10, 2023].

[54] Antonio SJ Musumeci. "MergerFS". *GitHub*. https://github.com/trapexit/mergerfs, 2014. [Accessed on Dec. 28, 2023].

[55] Chris Pavlou. "Why your Cassandra needs local NVMe and Rok". *Arrikto*. https://www.arrikto.com/tutorials/data-management/why-your-cassandra-needs-local-nvme-and-rok/, Oct. 2019. [Accessed on Jan. 08, 2024].

[56] Chris Pavlou. "Run Apache Cassandra on Kubernetes 15x Faster with Arrikto and Datastax". *Arrikto*. https://www.arrikto.com/tutorials/data-management/run-apache-cassandra-on-kubernetes-15x-faster-with-arrikto-and-datastax/, Jun. 2020. [Accessed on Jan. 08, 2024].

[57] Ubuntu. "What is Ceph?". *Ubuntu*. https://ubuntu.com/ceph/what-is-ceph. [Accessed on Jan. 08, 2024].

[58] ”Jie Yu, Saad Ali, James DeFelice, and the members of container-storage-interface-working-group@googlegroups.com”. ”Container Storage Interface (CSI) [v1.8.0]”. *GitHub*. https://github.com/container-storage-interface/spec/blob/v1.8.0/spec.md. [Accessed on Dec. 27, 2023].