ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επέκταση Συστήματος Αντιγράφων Ασφαλείας Εγγενούς στο Νέφος, με Χρήση Στιγμιοτύπων Δίσκων, για Διηπειρωτική Μετακίνηση Εφαρμογών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Θωμάς Γ. Παντελεάκος

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2025

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

Επέκταση Συστήματος Αντιγράφων Ασφαλείας Εγγενούς στο Νέφος, με Χρήση Στιγμιοτύπων Δίσκων, για Διηπειρωτική Μετακίνηση Εφαρμογών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Θωμάς Γ. Παντελεάκος

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Φεβρουαρίου 2025.

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

Αθήνα, Φεβρουάριος 2025

. . . . . .

Θωμάς Γ. Παντελεάκος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

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

Extension of a Cloud-Native Kubernetes Backup System for Cross-Region Restore With Disk Snapshots

DIPLOMA THESIS

Thomas G. Panteleakos

Supervisor: Nectarios Koziris
Professor NTUA

Athens, February 2025

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

Extension of a Cloud-Native Kubernetes Backup System for Cross-Region Restore With Disk Snapshots

DIPLOMA THESIS

Thomas G. Panteleakos

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the three-member examination committee on the 21st of February 2025.

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

Athens, February 2025

. . . . . .

Thomas G. Panteleakos
Electrical and Computer Engineer NTUA

στην οικογένειά μου

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.999996.

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

Περίληψη

Οι οργανισμοί υιοθετούν όλο και περισσότερο τον Κυβερνήτη για να βοηθήσουν στη διαχείριση των εφαρμογών τους με containerized τρόπο. Σε μεγάλη κλίμακα, αυτά τα περιβάλλοντα γίνονται εκθετικά πολύπλοκα, με την ανάγκη διατήρησης μόνιμων δεδομένων. Ως εκ τούτου, οι αξιόπιστες μέθοδοι δημιουργίας αντιγράφων ασφαλείας και επαναφοράς, οι οποίες μπορούν επίσης να δημιουργούν αντίγραφα ασφαλείας των δεδομένων των χρησιμοποιούμενων τόμων, είναι κρίσιμες. Έχουν αναπτυχθεί διάφορα εργαλεία για να βοηθήσουν στην αντιμετώπιση αυτής της ανάγκης, αλλά συχνά δεν ενσωματώνονται απρόσκοπτα με τα χρησιμοποιούμενα ειδικά συστήματα αποθήκευσης.

Η παρούσα διπλωματική εργασία επιχειρεί να γεφυρώσει το χάσμα μεταξύ ενός από τα πιο διαδεδομένα εργαλεία δημιουργίας αντιγράφων ασφαλείας και επαναφοράς, του Velero, και ενός δημοφιλούς συστήματος αποθήκευσης μπλοκ για τον Κυβερνήτη, του Longhorn. Ενώ κάποιες μέθοδοι μας βοηθούν ήδη να ενσωματώσουμε το Velero με ένα cluster που χρησιμοποιεί το Longhorn, δεν υπάρχει κάποιο ειδικό plugin για την ενσωμάτωση αυτών.

Με την ανάπτυξη ενός αποκλειστικού plugin του Velero για το Longhorn, βοηθάμε στην αυτοματοποίηση της δημιουργίας αντιγράφων ασφαλείας των στιγμιότυπων δίσκου του Longhorn και επεκτείνουμε τη δυνατότητά του να μεταφέρει επίσης αυτά τα δεδομένα σε έναν εξωτερικό κάδο S3, παρέχοντας μια κλιμακούμενη, cloud-native λύση αποκατάστασης καταστροφών. Αυτή η εργασία έχει ως στόχο να γεφυρώσει το χάσμα μεταξύ της διαχείρισης της αποθήκευσης του Κυβερνήτη και των λύσεων δημιουργίας αντιγράφων ασφαλείας που βασίζονται στο cloud. Συμβάλλει έτσι στην ανάπτυξη πιο ισχυρών και αυτοματοποιημένων μηχανισμών προστασίας δεδομένων σε περιβάλλοντα cloud-native, ενισχύοντας τόσο την αποδοτικότητα όσο και την ανθεκτικότητα των δεδομένων για τους χρήστες του Κυβερνήτη.

Λέξεις-Κλειδιά

Κυβερνήτης, Velero, Longhorn, Αποθήκευση σε Μπλοκ, Σταθεροί Τόμοι, Δημιουργία Αντιγράφων Ασφαλείας και Επαναφοράς, Αποκατάσταση Καταστροφών, Προστασία Δεδομένων, Containerized Εφαρμογές

Abstract

Organizations have increasingly adopted Kubernetes to help manage their applications in a containerized way. On a large scale, these environments become exponentially complex, with the need to hold persistent data. Therefore, reliable backup and restore methods, which can also back up the used volumes' data, are critical. Various tools have been developed to help tackle this need, but they often do not seamlessly integrate with the dedicated storage systems used.

This diploma thesis attempts to bridge the gap between one of the most widely used backup and restore tools, Velero, and a popular block storage system for Kubernetes, Longhorn. While some methods already help us integrate Velero with a cluster that utilizes Longhorn, there is no dedicated plugin to integrate these.

By developing a dedicated Velero plugin for Longhorn, we help automate the backup of Longhorn disk snapshots and extend its capability to also transfer this data to an external S3 bucket, providing a scalable, cloud-native disaster recovery solution. This work aims to bridge the gap between Kubernetes storage management and cloud-based backup solutions. It thus contributes to the development of more robust and automated data protection mechanisms in cloud-native environments, enhancing both operational efficiency and data resilience for Kubernetes users.

Keywords

Kubernetes, Velero, Longhorn, Block Storage, Persistent Volumes, Backup and Restore, Disaster Recovery, Data Protection, Containerized Applications

Αντί Προλόγου

Η διπλωματική αυτή εργασία αποτελεί το επιστέγασμα μιας προσπάθειας χρόνων, στην οποία έχουν συμβάλει πολλοί άνθρωποι. Πριν προχωρήσουμε στο κύριο μέρος της, θα ήθελα να αναφερθώ σε ορισμένους από αυτούς που βοήθησαν να ολοκληρωθεί επιτυχώς.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της εργασίας, κ. Νεκτάριο Κοζύρη, ο οποίος με την διδασκαλία του μου εμφύσησε το ενδιαφέρον για το αντικείμενο των Υπολογιστικών Συστημάτων.

Έπειτα, θα ήθελα να ευχαριστήσω θερμά τον Δρ. Βαγγέλη Κούκη. Από την πρώτη στιγμή που ξεκινήσαμε να συνεργαζόμαστε, ο ενθουσιασμός του για το αντικείμενο, η βαθιά του γνώση και η επιμονή του στην λεπτομέρεια θα είναι πολύτιμα μαθήματα που θα κρατήσω. Επίσης, θα ήθελα να τον ευχαριστήσω για την ευκαιρία να εκπονήσω την εργασία μου στα πλαίσια της εταιρίας Arrikto. Η δυνατότητα να συνεργαστώ και να αντιληφθώ ένα κομμάτι της καθημερινότητας μιας εταιρίας, θα αποτελεί πάντα ένα χρήσιμο δίδαγμα για εμένα.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και τους κοντινούς μου ανθρώπους για την αγάπη και την υποστήριξη σε όσα έχουν συμβεί μέχρι τώρα και σε όσα θα ακολουθήσουν.

Θωμάς Παντελεάκος
Φεβρουάριος 2025

Contents

Εκτενής Ελληνική Περίληψη

Παρακάτω παρατίθεται το κείμενο της παρούσας διπλωματικής εργασίας σε μορφή εκτενούς ελληνικής περίληψης.

1 Εισαγωγή

Σε αυτό το εναρκτήριο κεφάλαιο, θα περιγράψουμε την πορεία της εργασίας μας. Πρώτον, θα συζητήσουμε το πρόβλημα και τους λόγους για τους οποίους το αντιμετωπίζουμε. Στη συνέχεια, θα εξετάσουμε τις υπάρχουσες λύσεις προτού περιγράψουμε την προτεινόμενη λύση μας σε υψηλό επίπεδο. Τέλος, θα παρουσιάσουμε τη δομή της παρούσας διπλωματικής εργασίας.

1.1 Κίνητρο

Το Kubernetes είναι ένα από τα πιο σημαντικά έργα ανοικτού κώδικα τα τελευταία χρόνια, το οποίο διευκολύνει τον χρήστη να διαχειρίζεται containerized εφαρμογές σε κλίμακα, ενώ παράλληλα εκμεταλλεύεται τις εγγενείς στο νέφος αρχιτεκτονικές. Με αυτόν τον τρόπο, η δυναμική διαχείριση stateless και stateful εφαρμογών γίνεται ευκολότερη. Διάφορα εργαλεία έχουν δημιουργηθεί για να κάνουν ευκολότερη την ανάπτυξη ενός cluster Kubernetes. Ένα από τα πιο δημοφιλή είναι το K3S, το οποίο έχει σχεδιαστεί για να επιτρέπει την απρόσκοπτη εγκατάσταση μιας διανομής Kubernetes σε περιβάλλοντα με περιορισμένους πόρους ή ακόμη και σε συσκευές IoT. [1]

Ανάλογα με τη φύση της εφαρμογής, η διατήρηση δεδομένων μπορεί να είναι απαραίτητη. Κατά συνέπεια, οι οργανισμοί πρέπει να είναι σε θέση να διατηρούν αυτά τα δεδομένα ασφαλή, αναπτύσσοντας αποτελεσματικές στρατηγικές δημιουργίας αντιγράφων ασφαλείας και επαναφοράς. Με αυτόν τον τρόπο, μπορούν να επιτύχουν την ακεραιότητα και την ανακτησιμότητα, οι οποίες αποτελούν κρίσιμες αρχές στη διαχείριση δεδομένων. Σε αυτό το πλαίσιο, οι ισχυρές λύσεις δημιουργίας αντιγράφων ασφα-

λείας είναι κρίσιμες για την αποφυγή απώλειας δεδομένων και την ελαχιστοποίηση του χρόνου διακοπής λειτουργίας κατά τη διάρκεια βλαβών.

Ένας συχνά χρησιμοποιούμενος κανόνας για μια επιτυχημένη στρατηγική δημιουργίας αντιγράφων ασφαλείας και επαναφοράς είναι ο «3-2-1 backup rule». Σύμφωνα με αυτόν τον κανόνα, για να προστατεύσουμε αποτελεσματικά τα δεδομένα μας, πρέπει να έχουμε τρία αντίγραφα των δεδομένων μας αποθηκευμένα σε δύο διαφορετικούς τύπους μέσων, με ένα αντίγραφο να φυλάσσεται εκτός του χώρου μας. [2]

Παρόλο που ο κανόνας αυτός χρονολογείται από μια εποχή που δεν υπήρχαν οι σύγχρονες τεχνολογίες (όπως το cloud), πολλές φορές είναι συνετό να τον χρησιμοποιούμε ως κατευθυντήρια γραμμή για τον πλεονασμό και την ανάκτηση δεδομένων. Ως εκ τούτου, είναι ζωτικής σημασίας να ενσωματώνουμε την έννοια των αντιγράφων ασφαλείας που φυλάσσονται εκτός της τοποθεσίας μας. Τα αντίγραφα αυτά είναι μια μέθοδος δημιουργίας αντιγράφων ασφαλείας δεδομένων σε έναν απομακρυσμένο διακομιστή ή σε μέσα που μεταφέρονται εκτός του χώρου μας. [3]

Καθώς το Kubernetes επιτυγχάνει τη διατήρηση των δεδομένων χρησιμοποιώντας την έννοια των μόνιμων τόμων, το cluster πρέπει να περιλαμβάνει μια λύση αποθήκευσης για τη διαχείριση της δημιουργίας, του κύκλου ζωής και της διαγραφής των τόμων.Το Longhorn παρέχει αυτές τις δυνατότητες, μαζί με επιλογές για τη δημιουργία αντιγράφων ασφαλείας και την επαναφορά αυτών των τόμων. Ωστόσο, μαζί με τις ενσωματωμένες επιλογές που παρέχει το Longhorn για τη δημιουργία αντιγράφων ασφαλείας τόμων, έχουν αναπτυχθεί πολλαπλά εργαλεία που βοηθούν στη διαδικασία δημιουργίας αντιγράφων ασφαλείας και επαναφοράς ενός cluster.Ένα από τα πιο δημοφιλή είναι το Velero, το οποίο περιλαμβάνει επιλογές για τη δημιουργία αντιγράφων ασφαλείας και την επαναφορά τόσο των πόρων του Kubernetes όσο και των τόμων που περιέχουν τα δεδομένα των εφαρμογών. Αυτά τα εργαλεία προσφέρουν πολλές επιλογές για την αποκατάσταση μετά από καταστροφή και έχουν σχεδιαστεί για να παρέχουν τις δυνατότητές τους σε μια μεγάλη ποικιλία αρχιτεκτονικών clusters.

Το Velero στοχεύει να προσφέρει μια καθολική λύση στο ζήτημα της αποκατάστασης μετά από καταστροφή για clusters Kubernetes, αλλά αυτή η λύση δεν μπορεί να προσαρμοστεί άμεσα στις ανάγκες κάθε επιλογής ανάπτυξης. Το Velero έχει αναπτύξει ένα σύστημα plugins που συμβάλλει στη γεφύρωση αυτού του χάσματος και προσφέρει στους διαχειριστές clusters την ευκαιρία να έχουν ένα ειδικό plugin που να καλύπτει τις

ανάγκες κάθε χρησιμοποιούμενου συστήματος αποθήκευσης.

Για να αντιμετωπίσουμε την ανάγκη για off-site αντίγραφα ασφαλείας χρησιμοποιώντας το Longhorn, μπορούμε να εξερευνήσουμε τις επιλογές που μας δίνει το Velero για την ενσωμάτωση με άλλες πλατφόρμες. Με αυτόν τον τρόπο, θα υπάρχει μια άμεση διαδικασία που θα επιτρέπει την πραγματοποίηση αποτελεσματικών στρατηγικών δημιουργίας αντιγράφων ασφαλείας και επαναφοράς, χωρίς ο χρήστης να χρειάζεται να παρέχει πρόσθετες ρυθμίσεις στην συστάδα που θα ήταν διαφορετικά απαραίτητες.

1.2 Διατύπωση Προβλήματος

Όπως αναφέρθηκε παραπάνω, το Velero είναι ένα εργαλείο για την ασφαλή δημιουργία αντιγράφων ασφαλείας και επαναφορά clusters Kubernetes, την αποκατάσταση μετά από καταστροφή και τη μετεγκατάσταση πόρων και μόνιμων τόμων clusters Kubernetes. Το Velero χρησιμοποιεί plugins για κάθε σύστημα για να συνεργάζεται με την πληθώρα των λύσεων αποθήκευσης που προσφέρονται στο Kubernetes. Αν και υπάρχουν λύσεις για τα στιγμιότυπα τόμων ακόμη και στην περίπτωση των πλατφορμών αποθήκευσης που δεν διαθέτουν ειδικό plugin, συχνά είναι περιορισμένες σε δυνατότητες, συνέπεια στιγμιότυπων ή ταχύτητα.

Επιπλέον, για την εκτέλεση οποιασδήποτε πρόσθετης λογικής στην δημιουργία αντιγράφων ασφαλείας και επαναφοράς -όπως η μετακίνηση του αντιγράφου ασφαλείας σε μια εξωτερική λύση αποθήκευσης στο νέφος- ο χρήστης θα πρέπει να εκτελέσει χειροκίνητες ενέργειες, οι οποίες αυξάνουν την πολυπλοκότητα και τη διάρκεια αυτών των λειτουργιών. Λόγω αυτών, η διαδικασία δημιουργίας αντιγράφων ασφαλείας δεν είναι τόσο αποτελεσματική όσο χρειάζεται, ιδίως όσον αφορά την αυτοματοποίηση και την επεκτασιμότητα.

Για να μετριάσουμε αυτές τις ελλείψεις, μια λύση είναι η ανάπτυξη ενός νέου plugin με τις νέες λειτουργίες που χρειαζόμαστε. Με αυτόν τον τρόπο, μπορούμε να εισαγάγουμε μια μοναδική λύση για τη στρατηγική δημιουργίας αντιγράφων ασφαλείας και επαναφοράς χρησιμοποιώντας το Longhorn και το Velero. Αυτή η στρατηγική θα περιέχει όλες τις λειτουργίες που χρειαζόμαστε με όσο το δυνατόν λιγότερη ανθρώπινη παρέμβαση.

1.3 Υπάρχουσες Λύσεις

Το Velero, ένα από τα πιο συχνά χρησιμοποιούμενα εργαλεία δημιουργίας αντιγράφων ασφαλείας και επαναφοράς του Kubernetes, ενσωματώνεται με πολλούς παρόχους αποθηκευτικού χώρου εξαρχής μέσω του εκτεταμένου συστήματος plugins. Ωστόσο, μπορεί επίσης να υποστηρίξει Volume Snapshots με συστήματα αποθήκευσης που δεν διαθέτουν ειδικό plugin. Υπάρχουν διάφοροι τρόποι για να επιτευχθεί αυτό.

- CSI Snapshot using a generic CSI plugin
- File System Backup
- CSI Snapshot Data Movement

Παρά το γεγονός ότι αυτές οι τεχνικές είναι σε θέση να επιτύχουν τελικά τον στόχο μας για επιτυχή προστασία των δεδομένων σε clusters Kubernetes, όλες έχουν συγκεκριμένα μειονεκτήματα. Για παράδειγμα, το File System Backup δημιουργεί αντίγραφα ασφαλείας των δεδομένων από το τρέχον σύστημα αρχείων, οπότε τα δεδομένα δεν συλλαμβάνονται την ίδια χρονική στιγμή, οπότε είναι λιγότερο συνεπές από άλλες προσεγγίσεις των στιγμιότυπων. [4]

Αυτή η διαδικασία καθιστά τα αντίγραφα ασφαλείας μας λιγότερο συνεπή. Επιπλέον, αυτή η μέθοδος χρησιμοποιεί εξωτερικά εργαλεία δημιουργίας αντιγράφων ασφαλείας όπως το Restic και το Kopia, τα οποία λαμβάνουν στιγμιότυπα σε επίπεδο συστήματος αρχείων. Τέτοια αντίγραφα ασφαλείας τείνουν να είναι σημαντικά πιο αργά.

Στην περίπτωση του CSI snapshot, το Velero χρησιμοποιεί ένα γενικό plugin για το CSI. Αυτός ο τρόπος μπορεί επίσης να χρησιμοποιηθεί με το Longhorn, καθώς το Longhorn υποστηρίζει τον μηχανισμό στιγμιοτύπων CSI του Kubernetes. Ωστόσο, αυτή η λειτουργικότητα προσφέρει ελάχιστα χαρακτηριστικά, ώστε να μπορεί να ενσωματωθεί σε πολλαπλά συστήματα συμβατά με CSI. Ο χρήστης πρέπει επίσης να ενεργοποιήσει την υποστήριξη CSI στο cluster. Η υποστήριξη CSI είναι κάτι που δεν περιλαμβάνουν όλες οι διανομές Kubernetes. Για παράδειγμα, το K3S (η διανομή Kubernetes που χρησιμοποιήσαμε σε αυτή τη διπλωματική εργασία) δεν περιλαμβάνει το Common Snapshot Controller και τα σχετικά Snapshot CRDs. Ο χρήστης πρέπει να πραγματοποιήσει ενέργειες για να συμπεριλάβει αυτά τα στοιχεία στο cluster.

1.4 Προτεινόμενη Λύση

Το σύστημα plugins του Velero μας επιτρέπει να αναπτύσσουμε τα δικά μας plugins. Αυτά τα πρόσθετα μπορούν να είναι διαφόρων ειδών, όπως Object Store, Volume Snapshotter, Backup Item Action, Restore Item Action και Delete Item Action. Στην περίπτωσή μας, θα χρησιμοποιήσουμε τη διεπαφή Volume Snapshotter. Με αυτόν τον τρόπο, μπορούμε να δημιουργήσουμε ένα plugin που θα πετύχει το στόχο των off-site αντιγράφων ασφαλείας χρησιμοποιώντας το Longhorn.

Το plugin αποτελείται από ενέργειες που λαμβάνουν χώρα σε συγκεκριμένες χρονικές στιγμές. Πρώτον, ο χρήστης πρέπει να το εγκαταστήσει κατά την εγκατάσταση του Velero, οπότε πρέπει επίσης να δώσει ορισμένες παραμέτρους. Στη συνέχεια, όταν προσπαθεί να δημιουργήσει ένα νέο αντίγραφο ασφαλείας, το Velero όχι μόνο θα ενεργοποιήσει τη δημιουργία του αντιγράφου ασφαλείας του Longhorn, αλλά θα περιμένει να δημιουργηθεί και στη συνέχεια θα συγχρονίσει την Backupstore με έναν πρόσθετο εξωτερικό S3 bucket. Το Velero θα ενεργοποιήσει τη δημιουργία ενός τόμου Longhorn Volume κατά τη στιγμή της επαναφοράς με βάση το επιλεγμένο αντίγραφο ασφαλείας. Εάν συμβεί μια καταστροφή στο πρωτεύον S3 bucket όπου φιλοξενείται η Backupstore, ο χρήστης μπορεί εύκολα να ρυθμίσει την Backupstore ώστε να δείχνει στο εξωτερικό bucket. Στη συνέχεια, η διαδικασία επαναφοράς μπορεί να συνεχιστεί με τον ίδιο τρόπο όπως και πριν.

Αυτή η διαδικασία είναι όσο το δυνατόν πιο απλοποιημένη για Kubernetes/K3S clusters με μόνιμο αποθηκευτικό χώρο. Είναι επίσης επεκτάσιμη, καθώς ακόμη και σε πολλαπλά αντίγραφα ασφαλείας, ο χρήστης δεν χρειάζεται να μεταφέρει τα δεδομένα αντιγράφων ασφαλείας σε εξωτερικό διακομιστή με μη αυτόματο τρόπο.

Ως αποτέλεσμα, η παρούσα διπλωματική εργασία έχει ως στόχο να καλύψει ένα κενό στον τομέα της δημιουργίας αντιγράφων ασφαλείας και επαναφοράς του Kubernetes σε συστάδες που χρησιμοποιούν το Longhorn ως σύστημα αποθήκευσης. Προσπαθεί να απλοποιήσει και να αυτοματοποιήσει την πρακτική της αποστολής αντιγράφων ασφαλείας σε έναν εξωτερικό χώρο για τη διατήρηση καλών πρακτικών αποκατάστασης μετά από καταστροφή.

1.5 Δομή Διπλωματικής Εργασίας

Το υπόλοιπο της παρούσας διπλωματικής εργασίας διαρθρώνεται ως εξής.

- Στο **Κεφάλαιο 2**, παρέχουμε τις απαραίτητες βασικές πληροφορίες και το πλαίσιο για να μπορέσει ο αναγνώστης να παρακολουθήσει το θέμα της παρούσας διπλωματικής εργασίας.
- Στο **Κεφάλαιο 3**, εξηγούμε λεπτομερώς το σχεδιασμό των τρεχουσών λύσεων κατά τη χρήση του Velero για στιγμιότυπα τόμου.
- Στο **Κεφάλαιο 4**, παρέχουμε έναν λεπτομερή σχεδιασμό του Velero plugin για το Longhorn.
- Στο **Κεφάλαιο 5**, αξιολογούμε τις λύσεις μας με βάση μετρήσεις και σχολιάζουμε την αποτελεσματικότητά τους.
- Στο **Κεφάλαιο 6**, συνοψίζουμε τα ευρήματά μας, τις συνεισφορές μας και συζητάμε πιθανές μελλοντικές επεκτάσεις.

2 Υπόβαθρο

Σκοπός αυτού του κεφαλαίου είναι να εξηγήσει τις τεχνολογίες που χρησιμοποιήθηκαν για την παρούσα εργασία και τον τρόπο με τον οποίο ενσωματώνονται. Για να δοκιμάσουμε τις στρατηγικές δημιουργίας αντιγράφων ασφαλείας και επαναφοράς για τους τόμους κατά τη χρήση του Velero, χρειαζόμασταν ένα περιβάλλον δοκιμών που να είναι αναπαραγώγιμο, δωρεάν και αρκετά ελαφρύ ώστε να μπορεί να εγκατασταθεί σε έναν οικιακό υπολογιστή. Έτσι, στην συνέχεια περιγράφουμε τα χαρακτηριστικά κάθε ενός από τα εργαλεία που χρησιμοποιήσαμε και πώς τα αξιοποιήσαμε για να εξάγουμε τα συμπεράσματα που τελικά πήραμε. Η συνολική αρχιτεκτονική βασίζεται σε Virtualbox VMs, οπότε θα ξεκινήσουμε από εκεί. Το βασικό μας εργαλείο είναι το Kubernetes. Επομένως, πρέπει να εξηγήσουμε κάποιες βασικές έννοιες, ενώ η συγκεκριμένη διανομή Kubernetes που χρησιμοποιήσαμε είναι το K3S. Το σύστημα αποθήκευσης μπλοκ στο οποίο βασίσαμε το έργο μας είναι το Longhorn, και η αρχιτεκτονική εκμεταλλεύεται πολλά από τα χαρακτηριστικά του. Η πλατφόρμα αποθήκευσης αντικειμένων που θα φιλοξενεί τα εφεδρικά μας αρχεία είναι το MinIO. Τέλος, το κύριο εργαλείο που χρησιμοποιείται για την εκτέλεση όλων των εργασιών δημιουργίας αντιγράφων ασφαλείας και επαναφοράς είναι το Velero, συμπεριλαμβανομένου του αντιγράφου ασφαλείας και της επαναφοράς των μόνιμων τόμων.

2.1 VirtualBox

2.1.1 Εικονικές Μηχανές

Δεδομένου ότι θέλαμε να προσομοιώσουμε ένα περιβάλλον εργασίας σε ένα σύστημα, χρησιμοποιώντας έναν οικιακό υπολογιστή, ήταν απαραίτητο να χρησιμοποιήσουμε εικονικές μηχανές που θα χρησιμοποιούνταν ως κόμβοι του Kubernetes cluster. Μια Εικονική Μηχανή (VM) είναι ένας υπολογιστικός πόρος που χρησιμοποιεί λογισμικό αντί για φυσικό υλικό για την εκτέλεση προγραμμάτων και την ανάπτυξη εφαρμογών. [5] Αυτό είναι το εργαλείο που χρειαζόμασταν για να προσομοιώσουμε τους κόμβους μας. Υπάρχουν δύο τύποι εικονικών μηχανών: μια εικονική μηχανή διεργασίας και μια εικονική μηχανή συστήματος. Η εικονική μηχανή διεργασίας είναι μια εικονική πλατφόρμα που δημιουργείται για μια μεμονωμένη διεργασία και καταστρέφεται μόλις η διεργασία τερματιστεί, ενώ μια εικονική μηχανή συστήματος υποστηρίζει ένα λειτουργικό σύστημα μαζί με πολλές διεργασίες χρηστών. [6] Αυτό που χρειαζόμαστε είναι μια εικονική μηχανή συστήματος ώστε να μπορούμε να προσομοιώσουμε ένα πλήρες σύστημα. Η έννοια της Εικονικής Μηχανής συστήματος βασίζεται στην ύπαρξη ενός Hypervisor, δηλαδή του λογισμικού που διαχειρίζεται (με την έννοια ότι δημιουργεί, εκτελεί και καταστρέφει) όλες τις Εικονικές Μηχανές στο σύστημα. Υπάρχουν δύο τύποι Hypervisors:

- Hypervisor τύπου 1 (ή «bare metal»). Αυτός ο τύπος είναι ουσιαστικά ένα ελαφρύ λειτουργικό σύστημα που εκτελείται απευθείας στο υλικό του συστήματος.

- Hypervisor τύπου 2 (ή ”hosted”). Αυτός ο τύπος εκτελείται ως κανονική εφαρμογή πάνω από το λειτουργικό σύστημα και, επομένως, όλες οι κλήσεις συστήματος προς το υλικό του συστήματος περνούν μέσω του λειτουργικού συστήματος.

2.1.2 Προδιαγραφές των VM που χρησιμοποιούνται

Όπως αναφέρθηκε προηγουμένως, το VirtualBox είναι η πλατφόρμα που χρησιμοποιούμε για τη δημιουργία των VM που λειτουργούν ως κόμβοι του cluster K3S. Τα χαρακτηριστικά κάθε VM που χρησιμοποιείται ως κόμβος K3S βρίσκονται στον πίνακα 2.1.

Μνήμη	8192 MB
Επεξεργαστές	2
Εικονικός Σκληρός Δίσκος	200 GB
Προσαρμογείς Δικτύου	Bridged Network

Πίνακας 2.1: *Τα χαρακτηριστικά των VM που χρησιμοποιούνται για τους κόμβους K3S.*

Όσον αφορά τη δικτύωση, τα VM μας έχουν 1 κάρτα δικτύου το καθένα. Η κάρτα δικτύου έχει τη ρύθμιση Bridged Network. Με αυτήν τη δικτύωση, το VirtualBox χρησιμοποιεί ένα πρόγραμμα οδήγησης συσκευής που υπάρχει στο κεντρικό σύστημα, το οποίο φιλτράρει τα δεδομένα από τον φυσικό προσαρμογέα δικτύου. Αυτό το πρόγραμμα οδήγησης ονομάζεται πρόγραμμα οδήγησης net filter. Μόλις τα δεδομένα περάσουν από αυτόν τον οδηγό, το VirtualBox τα παραλαμβάνει και εισάγει δεδομένα, έτσι ώστε το σύστημα του κεντρικού υπολογιστή να ερμηνεύει το πακέτο που λαμβάνει σαν να προέρχεται από μια απευθείας συνδεδεμένη διασύνδεση δικτύου. Αυτό ουσιαστικά δημιουργεί μια νέα «κάρτα δικτύου λογισμικού».

Το λειτουργικό σύστημα που χρησιμοποιείται είναι Ubuntu Server 23.10 με έκδοση πυρήνα Linux 6.5.0-44-generic.

Στην περίπτωση της Εικονικής Μηχανής που χρησιμοποιείται για τη φιλοξενία του MinIO Object Storage, οι προδιαγραφές διαφέρουν λίγο. Αυτό οφείλεται στις διαφορετικές απαιτήσεις που έχει το λογισμικό που φιλοξενείται σε κάθε VM. Οι προδιαγραφές αυτού του VM φαίνονται στον πίνακα 2.2.

Μνήμη	4096 MB
Επεξεργαστές	2
Εικονικός Σκληρός Δίσκος	100 GB
Προσαρμογείς Δικτύου	Bridged Network

Πίνακας 2.2: *Τα χαρακτηριστικά των VM που χρησιμοποιούνται για τους κόμβους MinIO.*

Το λειτουργικό σύστημα και ο πυρήνας που χρησιμοποιούνται για τον κεντρικό υπολογιστή MinIO είναι τα ίδια με τους κεντρικούς υπολογιστές K3S: Ubuntu Server 23.10 με έκδοση πυρήνα Linux 6.5.0-44-generic.

2.2 Kubernetes

2.2.1 Βασικά στοιχεία του Kubernetes

Το Kubernetes είναι ένα σύστημα ενορχήστρωσης containers (CO). Τα συστήματα αυτά μπορούν να χρησιμοποιηθούν για την αυτόματη δημιουργία, ανάπτυξη, κλιμάκωση και διαχείριση εφαρμογών με container οπουδήποτε με ελάχιστη παρέμβαση στην υποκείμενη υποδομή. [7] Καθώς οι εφαρμογές που χρησιμοποιούν containers αυξάνονται σε πολυπλοκότητα και ποσότητα, η ανάγκη για ένα κεντρικό εργαλείο λογισμικού που είναι σε θέση να ενορχηστρώνει τις εργασίες, κατέστη σαφής. Το Kubernetes είναι ένα εργαλείο που μπορεί να καλύψει αυτή την ανάγκη, καθώς προγραμματίζει και αυτοματοποιεί τις εργασίες που σχετίζονται με τα containers καθ' όλη τη διάρκεια του κύκλου ζωής της εφαρμογής. [8]

Το Kubernetes μπορεί να αναπτυχθεί είτε on premises είτε σε μια πλατφόρμα cloud. Πολλές πλατφόρμες cloud προσφέρουν επίσης διαχειριζόμενες λύσεις Kubernetes, οι οποίες αναλαμβάνουν διάφορες εργασίες. Αυτές οι εργασίες μπορεί να επιφέρουν σημαντικό φόρτο στις ομάδες που τις διαχειρίζονται και μπορεί να περιλαμβάνουν τη διαμόρφωση απομονωμένων διαπιστευτηρίων, την αυτοανάκτηση, την εκτέλεση παρτίδων, τη διαχείριση φόρτου εργασίας, την προοδευτική ανάπτυξη εφαρμογών και άλλα. [9]

2.2.2 Αρχιτεκτονική του Kubernetes

Η ανάπτυξη του Kubernetes, σημαίνει τη δημιουργία ενός cluster από κόμβους. Ένα cluster Kubernetes αποτελείται από ένα σύνολο μηχανών, που ονομάζονται κόμβοι, οι οποίες εκτελούν εφαρμογές που περιέχουν containers. Κάθε cluster έχει τουλάχιστον έναν κόμβο worker. [10]

Ο κόμβος/οι κόμβοι εργασίας φιλοξενούν τα Pods που είναι τα συστατικά της λειτουργίας της εφαρμογής. Το Control Plane διαχειρίζεται τους κόμβους worker και τα Pods στο cluster.

2.3 K3S

Το K3S είναι μια ελαφριά διανομή Kubernetes, η οποία μπορεί να εγκατασταθεί σε μια μεγάλη ποικιλία περιβαλλόντων. Επιλέχθηκε, καθώς προσφέρει πολλαπλά πλεονεκτήματα, όπως το ότι είναι ελαφριά (το οποίο είναι σημαντικό σε ένα δοκιμαστικό περιβάλλον), η ευκολία εγκατάστασης και χρήσης και η υποστήριξη του Longhorn (η λύση αποθήκευσης που επιλέξαμε). Ένα πρόσθετο πλεονέκτημα του K3S είναι ότι φέρει προεγκατεστημένες τις απαιτούμενες εξαρτήσεις για πολλαπλές χρήσιμες λειτουργίες: [11]

- containerd / cri-dockerd container runtime (CRI)
- Flannel Container Network Interface (CNI)
- CoreDNS Cluster DNS
- Traefik Ingress controller
- ServiceLB Load-Balancer controller
- Kube-router Network Policy controller
- Local-path-provisioner Persistent Volume controller
- Spegel distributed container image registry mirror
- Host utilities (iptables, socat, etc)

2.4 Longhorn

Το Longhorn είναι μια cloud native λύση αποθήκευσης μπλοκ για το Kubernetes. Το Longhorn υλοποιεί κατανεμημένη αποθήκευση μπλοκ με τη χρήση containers και microservices. Δημιουργεί έναν αποκλειστικό ελεγκτή αποθήκευσης για κάθε τόμο μπλοκ συσκευής και αναπαράγει συγχρονισμένα τον όγκο σε πολλαπλά αντίγραφα που είναι αποθηκευμένα σε πολλαπλούς κόμβους. Αυτά τα αντίγραφα αποτελούνται από μια αλυσίδα στιγμιότυπων, παρουσιάζοντας το ιστορικό των αλλαγών στα δεδομένα εντός ενός τόμου. Ο ίδιος ο ελεγκτής αποθήκευσης και τα αντίγραφα ενορχηστρώνονται

χρησιμοποιώντας το Kubernetes. [12] Αυτό απλοποιεί την πολυπλοκότητα της κατανεμημένης αποθήκευσης, καθιστώντας ουσιαστικά κάθε τόμο το δικό του microservice. Ο ελεγκτής κάθε τόμου ονομάζεται *Longhorn Engine* και το συστατικό που ενορχηστρώνει όλα αυτά μαζί ονομάζεται *Longhorn Manager*.

Το Longhorn μας παρέχει έναν οδηγό Container Storage Interface (CSI) (*driver.longhorn.io*). Αυτός ο οδηγός παίρνει τη συσκευή μπλοκ, τη μορφοποιεί και την προσαρτά στον κόμβο. Στη συνέχεια, το kubelet προσαρτά τη συσκευή μέσα σε ένα Kubernetes Pod. Αυτό επιτρέπει στο Pod να έχει πρόσβαση στον τόμο Longhorn. [13]

2.5 MinIO

Το MinIO είναι ένα σύστημα αποθήκευσης αντικειμένων υψηλής απόδοσης. Είναι εγγενές στο Kubernetes και μπορεί να χρησιμοποιηθεί για την παροχή λύσεων διαχείρισης δεδομένων σε εφαρμογές που χρησιμοποιούν το Kubernetes. Είναι πλήρως συμβατό με το Amazon S3, επομένως μπορεί να χρησιμεύσει ως μια δωρεάν και ανοικτού κώδικα εναλλακτική λύση για περιβάλλοντα ανάπτυξης και δοκιμών. Διατίθεται σε διάφορες εκδόσεις, συμπεριλαμβανομένων επιλογών ανάπτυξης για Kubernetes, Docker, Linux, MacOS και Windows. Το MinIO μπορεί να αναπτυχθεί είτε σε αυτόνομη είτε σε κατανεμημένη διαμόρφωση. [14]

Το MinIO προσφέρει πολλαπλά χαρακτηριστικά, μερικά από τα οποία είναι η κωδικοποίηση διαγραφής (erasure coding), η προστασία bitrot και η κρυπτογράφηση.

2.6 Velero

Το Velero είναι ένα εργαλείο ανοιχτού κώδικα που επιτρέπει στους χρήστες να δημιουργούν αντίγραφα ασφαλείας και να επαναφέρουν τους πόρους του Kubernetes cluster και τους μόνιμους τόμους. [15] Αποτελείται από έναν διακομιστή που εκτελείται στο cluster και έναν client γραμμής εντολών που εκτελείται τοπικά.

2.6.1 Μηχανισμός plugins του Velero

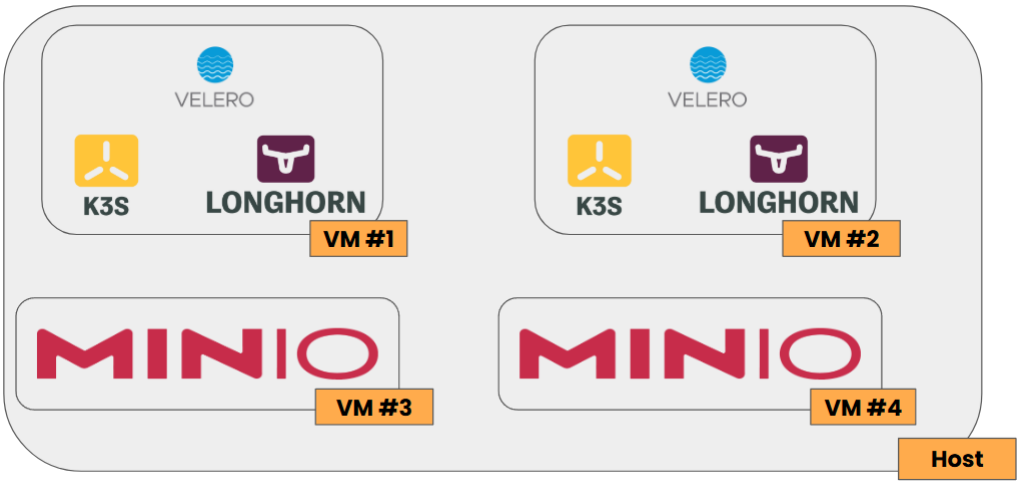
Το Velero χρησιμοποιεί ένα σύστημα plugins για την αλληλεπίδραση με πολλαπλά συστήματα αποθήκευσης. Οι χρήστες μπορούν να προσθέσουν τα δικά τους plugins για να υλοποιήσουν τη δική τους προσαρμοσμένη λειτουργικότητα στα αντίγραφα ασφαλείας & στις διαδικασίες επαναφοράς του Velero χωρίς να χρειάζεται να τροποποιήσουν/επαναμεταγλωττίσουν το βασικό αρχείο του Velero. Οι χρήστες μπορούν να δημιουργήσουν το δικό τους image που περιέχει οποιεσδήποτε επιπλέον λειτουργίες και αυτό το image αναπτύσσεται ως init container για το Velero server pod και αντιγράφει το αρχείο σε έναν κοινόχρηστο τόμο emptyDir για να έχει πρόσβαση ο Velero server.

Τα πρόσθετα μπορούν να είναι ενός από τους ακόλουθους τύπους:

- Object Store: Διατηρεί και ανακτά αντίγραφα ασφαλείας, αρχεία καταγραφής αντιγράφων ασφαλείας και αρχεία καταγραφής επαναφοράς.
- Volume Snapshotter: Δημιουργεί στιγμιότυπα τόμων (κατά τη δημιουργία αντιγράφων ασφαλείας) και επαναφέρει τόμους από στιγμιότυπα (κατά την επαναφορά).
- Backup Item Action: Εκτελεί αυθαίρετη λογική για μεμονωμένα στοιχεία πριν από την αποθήκευσή τους σε ένα αρχείο αντιγράφων ασφαλείας.
- Restore Item Action: Εκτελεί αυθαίρετη λογική για μεμονωμένα στοιχεία πριν από την επαναφορά τους σε ένα cluster.
- Delete Item Action: Εκτελεί αυθαίρετη λογική με βάση μεμονωμένα στοιχεία σε ένα αντίγραφο ασφαλείας πριν από τη διαγραφή του αντιγράφου ασφαλείας.

2.7 Αρχιτεκτονική συστήματος

Όλα τα παραπάνω εργαλεία χρησιμοποιήθηκαν για τη δημιουργία ενός χώρου εργασίας δοκιμών. Η πλήρης αρχιτεκτονική του συστήματος και ο τρόπος με τον οποίο ενσωματώνονται μεταξύ τους, φαίνεται στο παρακάτω διάγραμμα.



Σχήμα 2.1: *Η Αρχιτεκτονική του συστήματος.*

3 Σχεδίαση

Η διαδικασία δημιουργίας ενός αντιγράφου ασφαλείας σε ένα περιβάλλον Kubernetes που χρησιμοποιεί τόμους Longhorn με τη χρήση του Velero μπορεί να λάβει πολλαπλές μορφές. Σε αυτό το κεφάλαιο, θα εξετάσουμε τις διάφορες διαθέσιμες επιλογές, με βάση τα καθοριστικά χαρακτηριστικά της καθεμιάς.

3.1 Διαφορετικοί Τύποι Αντιγράφων Ασφαλείας

Το Velero μας επιτρέπει να δημιουργούμε αντίγραφα ασφαλείας για τα clusters, συμπεριλαμβανομένων των τόμων που είναι συνδεδεμένα με τα pods. Όταν χρησιμοποιούμε την παραπάνω αρχιτεκτονική, το Velero χρησιμοποιεί τρεις μηχανισμούς για να διατηρήσει την κατάσταση των volumes.

- Στιγμιότυπο CSI (CSI Snapshot)
- Αντίγραφο Ασφαλείας Συστήματος Αρχείων (File System Backup)
- Μεταφορά Δεδομένων Στιγμιότυπου CSI (CSI Snapshot Data Movement)

Στις επόμενες υποενότητες, θα αναφέρουμε επιγραμματικά πώς λειτουργεί εσωτερικά κάθε μέθοδος, πώς μπορούμε να την υλοποιήσουμε και να την ενσωματώσουμε στο cluster μας.

3.1.1 CSI Snapshot

Το CSI Snapshot αξιοποιεί την υποστήριξη του CSI (Container Storage Interface) στο Velero. Το CSI είναι ένα πρότυπο για την έκθεση συστημάτων αποθήκευσης μπλοκ και αρχείων σε containerized εφαρμογές σε Συστήματα Ορχήστρωσης Containers (COs) όπως το Kubernetes. Χρησιμοποιώντας το CSI, οι πάροχοι αποθήκευσης συστημάτων μπορούν να γράψουν και να αναπτύξουν plugins που εκθέτουν νέα συστήματα αποθήκευσης στο Kubernetes χωρίς να χρειάζεται να τροποποιήσουν τον βασικό κώδικα του Kubernetes. [16]

Όπως αναφέρθηκε στο Κεφάλαιο 2, το Longhorn χρησιμοποιεί επίσης το CSI. Το CSI plugin του Longhorn μας επιτρέπει να δημιουργούμε στιγμιότυπα των τόμων μας, και αυτά τα στιγμιότυπα μπορεί να είναι είτε Longhorn Snapshots, Longhorn Backups και Longhorn Backing Images.

3.1.2 File System Backup

Το Velero διαθέτει έναν μηχανισμό για τη δημιουργία αντιγράφων ασφαλείας και την επαναφορά τόμων Kubernetes που είναι συνδεδεμένοι με pods από το σύστημα αρχείων των τόμων. Αυτό ονομάζεται File System Backup (FSB) ή PodVolumeBackup. Η χρήση του είναι να επιτρέπει τη δημιουργία αντιγράφων ασφαλείας τόμων που δεν διαθέτουν ενσωματωμένο μηχανισμό δημιουργίας αντιγράφων ασφαλείας. Αυτή η δυνατότητα του Velero παρέχεται μέσω της ενσωμάτωσης δύο εργαλείων ανοιχτού κώδικα για δημιουργία αντιγράφων ασφαλείας: του restic και του kopia.

Το Αντίγραφο Ασφαλείας Συστήματος Αρχείων αποτελεί μια επιπρόσθετη λύση στην υπάρχουσα μέθοδο των CSI Snapshots, τα οποία αξιοποιούν τον CSI driver κάθε πλατφόρμας αποθήκευσης.

3.1.3 CSI Snapshot Data Movement

Το CSI Snapshot Data Movement είναι μια εναλλακτική μέθοδος για τη δημιουργία αντιγράφων ασφαλείας. Το Velero έχει αναπτύξει έναν νέο σχεδιασμό, όπου αρχικά λαμβάνεται ένα Στιγμιότυπο CSI και στη συνέχεια αποκτά πρόσβαση στα δεδομένα του στιγμιότυπου μέσω επιλογής Εργαλείων Μεταφοράς Δεδομένων (Data Movers), ώστε

να δημιουργήσει αντίγραφο ασφαλείας των δεδομένων σε μια αποθήκη αντιγράφων ασφαλείας συνδεδεμένη με τα Εργαλεία Μεταφοράς Δεδομένων. [17] Η Μεταφορά Δεδομένων Στιγμιότυπου CSI μπορεί να είναι χρήσιμη στις ακόλουθες περιπτώσεις:

- Για χρήστες on-prem: Πολλές λύσεις αποθήκευσης on-prem δεν προσφέρουν ανθεκτικά στιγμιότυπα. Για παράδειγμα, τα Δεδομένα Στιγμιότυπου Longhorn αποθηκεύονται στο cluster. Χρησιμοποιώντας τη δυνατότητα Μεταφοράς Δεδομένων Στιγμιότυπου CSI, ο χρήστης μπορεί πρώτα να λάβει ένα Στιγμιότυπο CSI και στη συνέχεια να μεταφέρει τα δεδομένα σε μια διαφορετική/φθηνότερη-/πιο ανθεκτική αποθήκευση.

- Για χρήστες δημόσιου cloud: Η λύση Μεταφοράς Δεδομένων μπορεί να αξιοποιηθεί για την εκτέλεση μιας στρατηγικής πολλαπλών cloud. Ο χρήστης μπορεί να πάρει ένα στιγμιότυπο χρησιμοποιώντας τους μηχανισμούς στιγμιοτύπων ενός παρόχου και στη συνέχεια να χρησιμοποιήσει ένα Εργαλείο Μεταφοράς Δεδομένων για να μεταφέρει αυτά τα δεδομένα σε έναν διαφορετικό πάροχο. Αυτό επιτρέπει στο Velero να λειτουργεί ως το κύριο εργαλείο δημιουργίας αντιγράφων ασφαλείας και επαναφοράς, ακόμη και σε περιβάλλον πολλαπλών cloud.

Ο σχεδιασμός CSI Snapshot Data Movement στο Velero υποστηρίζει τόσο έναν ενσωματωμένο εργαλείο μεταφοράς δεδομένων όσο και προσαρμοσμένα εργαλεία μεταφοράς δεδομένων. Το Ενσωματωμένο Εργαλείο Μεταφοράς Δεδομένων του Velero (VBDM) χρησιμοποιεί εργαλεία που είναι ήδη ενσωματωμένα από τον σχεδιασμό του Αντιγράφου Ασφαλείας Συστήματος Αρχείων: το Kopia Uploader και το Kopia Repository. Έτσι, το VBDM χρησιμοποιείται για τη λήψη στιγμιότυπου από ένα `PersistentVolume` και στη συνέχεια για την αποστολή των αρχείων του στιγμιότυπου μέσω του Kopia Uploader στο Kopia Repository. Αυτό ουσιαστικά πραγματοποιείται και με τη χρήση του File System Backup (με το Kopia ή το Restic ως πάροχο), αλλά με τον σχεδιασμό Μεταφοράς Δεδομένων:

1. Το Velero δημιουργεί ένα Στιγμιότυπο CSI του PersistentVolume χρησιμοποιώντας έναν οδηγό CSI.
2. Το Ενσωματωμένο Εργαλείο Μεταφοράς Δεδομένων του Velero (VBDM) δημιουργεί έναν τόμο από το στιγμιότυπο CSI.

3. Το VBDM μεταφέρει τα δεδομένα στην BackupStorageLocation χρησιμοποιώντας το Kopia.
4. Όταν ολοκληρωθεί η μεταφορά των δεδομένων ή παρουσιαστεί κάποιο σφάλμα, το VBDM θέτει το DataUpload CR (ένα νέο CR που έχει εισαχθεί) σε τερματική κατάσταση, είτε Completed είτε Failed.

3.2 Διαδικασία Restore με το Velero

Χρησιμοποιώντας τη δυνατότητα Restore, ο χρήστης μπορεί να χρησιμοποιήσει το Velero για να επαναφέρει όλους τους πόρους Kubernetes και τους τόμους που περιλαμβάνονταν σε ένα αντίγραφο ασφαλείας που είχε δημιουργηθεί προηγουμένως, ανεξαρτήτως της μεθόδου με την οποία είχε ληφθεί το αντίγραφο ασφαλείας. Εναλλακτικά, το Velero παρέχει στον χρήστη τη δυνατότητα να φιλτράρει τους πόρους που θα επαναφερθούν, με βάση διάφορες παραμέτρους, όπως το namespace, ο τύπος πόρου και οι ετικέτες επιλογής. Υπάρχει επίσης η επιλογή να αντιστοιχίσει ξανά το namespace στο οποίο ανήκουν οι πόροι ενός namespace, επαναφέροντάς τους σε ένα νέο namespace.

4 Υλοποίηση

Στα προηγούμενα κεφάλαια, συζητήσαμε το κίνητρο για τη δημιουργία ενός Longhorn-native plugin για το Velero, θέσαμε τις βάσεις γνώσεων για την ανάπτυξη αυτού του plugin και αναφέραμε τους εσωτερικούς μηχανισμούς όλων των υπαρχουσών λύσεων για τη δημιουργία αντιγράφων ασφαλείας ενός τόμου Longhorn χρησιμοποιώντας το Velero. Σε αυτό το κεφάλαιο, θα παραθέσουμε συνοπτικά τις λεπτομέρειες υλοποίησης και ανάπτυξης του plugin που δημιουργήσαμε.

4.1 Επισκόπηση

Όπως αναφέρθηκε στο Κεφάλαιο 2.6.1, το Velero χρησιμοποιεί plugins για να αλληλεπιδρά με διαφορετικές πλατφόρμες αποθήκευσης. Αυτά τα plugins μπορεί να είναι διαφόρων τύπων, όπως Object Store, Volume Snapshotter, Backup Item Action, Restore Item Action και Delete Item Action. Για τους σκοπούς της διπλωματικής μας εργασίας, εστιάσαμε στην υλοποίηση ενός plugin τύπου Volume Snapshotter.

4.2 Υλοποίηση

4.2.1 Διεπαφή VolumeSnapshotter του Velero

Στη διπλωματική αυτή εργασία, στόχος ήταν η ανάπτυξη ενός plugin που χειρίζεται και προσθέτει επιπλέον βήματα στα Στιγμιότυπα Τόμου (Volume Snapshots). Το Velero

μας παρέχει τη διεπαφή VolumeSnapshotter γι' αυτόν τον σκοπό, την οποία πρέπει να υλοποιήσει το plugin μας. Αυτή η διεπαφή ορίζει τις λειτουργίες που χρειάζεται το Velero για τη λήψη στιγμιότυπων των μόνιμων τόμων κατά τη δημιουργία αντιγράφων ασφαλείας και την επαναφορά των μόνιμων τόμων από στιγμιότυπα κατά την επαναφορά. [18]

```
type VolumeSnapshotter interface {
    Init(config map[string]string) error

    CreateVolumeFromSnapshot(snapshotID, volumeType, volumeAZ string, iops *int64) (volumeID ↩
        string, err error)

    GetVolumeID(pv runtime.Unstructured) (string, error)

    SetVolumeID(pv runtime.Unstructured, volumeID string) (runtime.Unstructured, error)

    GetVolumeInfo(volumeID, volumeAZ string) (string, *int64, error)

    CreateSnapshot(volumeID, volumeAZ string, tags map[string]string) (snapshotID string, err ↩
        error)

    DeleteSnapshot(snapshotID string) error
}
```

Listing 4.1: *Η διεπαφή VolumeSnapshotter.*

Κάθε λειτουργία πρέπει να υλοποιηθεί από μια συνάρτηση, και η εργασία που εκτελεί περιγράφεται παρακάτω:

- `Init`: προετοιμάζει τον VolumeSnapshotter για χρήση, χρησιμοποιώντας το παρεχόμενο map ζευγών κλειδιών-τιμών ρύθμισης. Επιστρέφει σφάλμα εάν ο Volume Snapshotter δεν μπορεί να αρχικοποιηθεί από τη συγκεκριμένη ρύθμιση.
- `CreateVolumeFromSnapshot`: δημιουργεί έναν νέο τόμο στη συγκεκριμένη ζώνη διαθεσιμότητας, αρχικοποιημένο από το παρεχόμενο στιγμιότυπο, με τον καθορισμένο τύπο και IOPS (εάν χρησιμοποιείται παρεχόμενο IOPS).
- `GetVolumeID`: επιστρέφει το ειδικό αναγνωριστικό του παρόχου cloud για τον Persistent Volume.
- `SetVolumeID`: ορίζει το ειδικό αναγνωριστικό του παρόχου cloud για τον Persistent Volume.
- `GetVolumeInfo`: επιστρέφει τον τύπο και το IOPS (εάν χρησιμοποιείται παρεχόμενο IOPS) για τον συγκεκριμένο τόμο στη δεδομένη availability zone.
- `CreateSnapshot`: δημιουργεί ένα στιγμιότυπο του συγκεκριμένου τόμου και εφαρμόζει το παρεχόμενο σύνολο ετικετών στο στιγμιότυπο.
- `DeleteSnapshot`: διαγράφει το συγκεκριμένο στιγμιότυπο τόμου.

Δεν είναι απαραίτητο όλες αυτές οι μέθοδοι να υλοποιούνται με τον ίδιο τρόπο για κάθε

plugin, καθώς οι διαφορετικές λύσεις αποθήκευσης έχουν διαφορετικές δυνατότητες και κάθε plugin μπορεί να ρυθμιστεί ώστε να έχει διαφορετικό αποτέλεσμα.

4.2.2 Υλοποίηση του Plugin Velero για Longhorn

Στην υλοποίησή μας, διαμορφώσαμε το αρχείο `main.go` ώστε να καταχωρεί ένα plugin τύπου VolumeSnapshotter.

Η λογική του plugin μας περιλαμβάνεται στο αρχείο `volumesnapshotterplugin.go`. Αυτό το αρχείο περιέχει τις υλοποιήσεις όλων των συναρτήσεων που απαιτούνται για ένα plugin τύπου VolumeSnapshotter.

4.3 Deployment

4.3.1 Deployment του Plugin

Όπως αναφέρθηκε παραπάνω, τα plugins του Velero υλοποιούν το σύστημα go-plugin. Στο πλαίσιο του Velero, αυτά είναι containers που αποτελούν μέρος του κύριου pod του Velero, το οποίο βρίσκεται στο namespace του Velero. Επομένως, για να χρησιμοποιήσουμε αυτό το προσαρμοσμένο plugin, πρέπει πρώτα να δημιουργήσουμε την εικόνα (image) και στη συνέχεια να την κάνουμε deploy. Για αυτόν τον σκοπό, μπορούμε να χρησιμοποιήσουμε το Makefile που παρέχεται από το Velero.

Χρησιμοποιώντας την εντολή `IMAGE=<repo>/<name> VERSION=<tag> make container`, μπορούμε να δημιουργήσουμε την εικόνα. Αυτή η εντολή καλεί το `docker build -t $(IMAGE):$(VERSION) .`. Στη συνέχεια, πρέπει να την ανεβάσουμε σε ένα registry που είναι προσβάσιμο από το cluster μας. Για αυτόν τον σκοπό, χρησιμοποιήσαμε το Dockerhub και ανεβάσαμε τις εικόνες χρησιμοποιώντας την εντολή `docker push $(IMAGE):$(VERSION)`. Τώρα που η εικόνα μας έχει δημιουργηθεί και είναι διαθέσιμη σε ένα προσβάσιμο registry, πρέπει να προσθέσουμε το plugin στην εγκατάσταση του Velero. Αυτό μπορεί να γίνει είτε κατά τη διάρκεια της εγκατάστασης του Velero είτε αφού το Velero έχει ήδη εγκατασταθεί.

4.3.2 Δημιουργία των απαραίτητων πόρων

Για να εκτελέσει το plugin όλες τις λειτουργίες του σωστά, πρέπει να ρυθμιστούν συγκεκριμένες τιμές. Αυτές είναι οι εξής:

- Backup Target URL: [19] Αυτό είναι το endpoint του Backupstore. Η μορφή του είναι `s3:/<bucketname>@<region>/`. Στην περίπτωσή μας, όπου το Backupstore είναι ένας διακομιστής MinIO, χρησιμοποιούμε `s3://backupbucket@us-east-1/`.

- Backup Target Secret: Αυτό είναι ένας πόρος Secret που ο χρήστης πρέπει να προσθέσει στο Kubernetes cluster. Βρίσκεται στο namespace `longhorn-system` και περιέχει όλες τις απαραίτητες πληροφορίες για το Backupstore, όπως το ACCESS_KEY_ID, το SECRET_ACCESS_KEY, το ENDPOINT και αν χρησιμοποιεί SSL. Όλες αυτές οι τιμές πρέπει να προστεθούν κωδικοποιημένες σε base64.

- Endpoint του Longhorn: Αυτό είναι το endpoint (διεύθυνση IP) του διακομιστή API του Longhorn. Σε μια τοπική εγκατάσταση του Longhorn, μπορεί να είναι η διεύθυνση IP του master node.

- Destination Secret: Αυτό είναι ένα Secret που περιέχει όλες τις απαραίτητες πληροφορίες για το cluster στο οποίο θα συγχρονιστούν τα δεδομένα του τόμου. Πρέπει να περιλαμβάνει τα ίδια πεδία με το Μυστικό Στόχου Αντιγράφου Ασφαλείας.

Εάν ο χρήστης θέλει να αποκρύψει τα ονόματα των πόρων, έχει τη δυνατότητα να δημιουργήσει ένα αρχείο και να χρησιμοποιήσει μεταβλητές περιβάλλοντος.

Συνοψίζοντας, οι χρήστες πρέπει να κάνουν τα εξής για να χρησιμοποιήσουν το plugin σε ένα cluster:

1. Δημιουργία των απαραίτητων πόρων.

2. Δημιουργία ενός αρχείου credentials.

3. Προσθήκη των μεταβλητών περιβάλλοντος.

4. Προσθήκη του plugin (είτε εγκαθιστώντας το Velero από την αρχή είτε εφαρμόζοντας τη μέθοδο της επί τόπου εγκατάστασης που αναφέρθηκε παραπάνω).

5. Επεξεργασία του VolumeSnapshotLocation για να χρησιμοποιήσει το plugin ως τον πάροχο στιγμιότυπων (snapshot).

4.3.3 Ανάκτηση από το backup bucket

Μετά την εγκατάσταση του plugin, κάθε φορά που ο χρήστης παίρνει αντίγραφο ασφαλείας του cluster χρησιμοποιώντας το Velero, τα δεδομένα του τόμου δεν θα πηγαίνουν μόνο στην Backupstore, αλλά θα συγχρονίζονται επίσης σε άλλο bucket (ας το ονομάσουμε backup bucket). Με αυτόν τον τρόπο, εάν το bucket που χρησιμοποιείται ως Backupstore καταστεί για οποιονδήποτε λόγο μη διαθέσιμο, ο χρήστης μπορεί να ρυθμίσει το Backup Target ώστε να δείχνει στο backup bucket και όλα τα αντίγραφα ασφαλείας θα είναι ξανά διαθέσιμα.

Για να το κάνει αυτό, ο χρήστης μπορεί να χρησιμοποιήσει το Longhorn UI και να πλοηγηθεί στο `Setting` -> `General`. Εκεί θα βρει την ρύθμιση που ονομάζεται `Backup Target Credential Secret` και μπορεί να την επεξεργαστεί ώστε να ταιριάζει με το όνομα του secret που προστέθηκε για το backup bucket.

Εάν πλοηγηθεί στη σελίδα `Backup` του Longhorn UI, θα βρει όλα τα αντίγραφα ασφαλείας διαθέσιμα στο backup bucket. Στην συνέχεια μπορεί να προχωρήσει σητν επαναφορά του αντιγράφου ασφαλείας με τον ίδιο τρόπο όπως θα προχωρούσε αρχικά.

5 Αξιολόγηση

Σε αυτό το κεφάλαιο, παρουσιάζουμε μια αξιολόγηση του σχεδιασμού και της υλοποίησής μας, παρέχοντας μερικούς δείκτες απόδοσης. Για να το κάνουμε αυτό, θα εξετάσουμε όλες τις μεθόδους που είναι δυνατές για τη δημιουργία αντιγράφου ασφαλείας χρησιμοποιώντας το Velero.

5.1 Δημιουργία Benchmark

Για να μπορέσουμε να δοκιμάσουμε όλες τις μεθόδους ακολουθώντας τις ίδιες διαδικασίες, δημιουργήσαμε ένα bash script που μετρά την απόδοση των αντιγράφων ασφαλείας του Velero για διάφορους τύπους (CSI, File System Backup και CSI Snapshot Data Movement), καθώς και για διάφορα plugins (CSI plugin, το δικό μας custom Velero plugin για Longhorn) με τα εξής βήματα:

1. Δημιουργία ενός test pod με έναν Longhorn τόμο συνδεδεμένο.
2. Δημιουργία ενός καθορισμένου από τον χρήστη ποσού δεδομένων μέσα σε αυτόν τον τόμο.
3. Δημιουργία αντιγράφων ασφαλείας με το Velero σε διάφορα στάδια (αρχικό, τροποποιημένα αρχεία, νέα αρχεία), για διάφορα σύνολα δεδομένων και χρησιμοποιώντας διαφορετικές μεθόδους δημιουργίας αντιγράφων ασφαλείας τόμου.
4. Καταγραφή χρονικών σημείων για τον υπολογισμό της διάρκειας του αντιγράφου ασφαλείας για κάθε στάδιο και τύπο.

Στην συνέχεια παραθέτουμε την ακολουθία του script. Εκτελεί τα εξής:

1. Ορίζει τη χρήση του.
2. Εκτελεί δοκιμές επικύρωσης για να εξασφαλίσει σωστή χρήση.
3. Δημιουργεί τρεις πόρους στο cluster: ένα namespace, ένα PersistentVolumeClaim και ένα pod που χρησιμοποιεί αυτό το PersistentVolumeClaim.
4. Αναπτύσσει ένα υπο-script για να δημιουργήσει τα τυχαία αρχεία που ζητήθηκαν μέσα στον νέο Longhorn τόμο.
5. Εκτελεί το ζητούμενο αντίγραφο ασφαλείας.
6. Προσθέτει την έξοδο της εντολής `velero describe` για αυτό το αντίγραφο ασφαλείας σε ένα αρχείο, περιλαμβάνοντας μετρικές για την εκτέλεση αυτού του script.
7. Προσθέτει τα χρονικά σημεία του αντιγράφου ασφαλείας και τη συνολική διάρκεια του αντιγράφου ασφαλείας στο αρχείο.
8. Εκτελεί μια λειτουργία touch σε όλα τα δημιουργημένα αρχεία χρησιμοποιώντας ένα υπο-script.
9. Παίρνει ένα επιπλέον αντίγραφο ασφαλείας και προσθέτει την αντίστοιχη έξοδο στο αρχείο με τις μετρικές.
10. Δημιουργεί 30000 επιπλέον αρχεία στον τόμο.
11. Παίρνει ακόμη ένα αντίγραφο ασφαλείας και προσθέτει την αντίστοιχη έξοδο στο αρχείο με τις μετρικές.
12. Διαγράφει τους δημιουργημένους πόρους.

Όπως αναφέρθηκε παραπάνω, αυτό το script χρησιμοποιεί κάποια υπο-scripts, κυρίως για να δημιουργήσει τα ζητούμενα αρχεία μέσα στον Longhorn τόμο. Αυτά τα υπο-scripts είναι τα εξής:

- **create-files.sh**: Δημιουργεί τυχαία αρχεία μέσα στον τόμο, τον αριθμό και το μέγεθος των οποίων καθορίζει ο χρήστης κατά την εκκίνηση του script.
- **touch-files.sh**: Εκτελεί μια λειτουργία `touch` σε όλα τα αρχεία που δημιουργήθηκαν προηγουμένως. Αυτό δοκιμάζει τη συμπεριφορά του αντιγράφου ασφαλείας αφού τα αρχεία έχουν αποκτηθεί (αλλάζουμε τα χρονικά τους σημεία).
- **inc-files.sh**: Προσθέτει 30000 νέα αρχεία με μέγεθος 1KB. Αυτό έχει αποφασιστεί αυθαίρετα, για να επιδειχθούν οι δυνατότητες του αντιγράφου ασφαλείας μετά τη δημιουργία νέων αρχείων.

5.2 Αποτελέσματα

Για να δοκιμάσουμε τους μηχανισμούς στιγμιότυπων σε διάφορες συνθήκες, θα ορίσουμε τρία σύνολα δεδομένων, το καθένα από τα οποία θα αποτελείται από περίπου 10GB δεδομένων.

- Το σύνολο δεδομένων **large-file** θα περιλαμβάνει 1 αρχείο με μέγεθος 10GB.
- Το σύνολο δεδομένων **medium-file** θα περιλαμβάνει 1000 αρχεία, το καθένα με μέγεθος 10MB.
- Το σύνολο δεδομένων **small-file** θα περιλαμβάνει 625000 αρχεία, το καθένα με μέγεθος 16KB.

Δοκιμάσαμε αυτά τα σύνολα δεδομένων που περιέχουν τυχαία δεδομένα ενάντια στους περιγραφόμενους μηχανισμούς αντιγράφου ασφαλείας: CSI Snapshot χρησιμοποιώντας το κοινό plugin CSI, CSI Snapshot χρησιμοποιώντας το Velero Plugin για το Longhorn, File System Backup (με το `kopia` ως πάροχο) και Velero CSI Snapshot Data Movement. Το CSI snapshot χρησιμοποιώντας το plugin CSI ρυθμίστηκε ώστε να δημιουργεί Longhorn Backups.

5.2.1 Παρατηρήσεις

Εξετάζοντας τα benchmarks που αναλύθηκαν παραπάνω, μπορούμε να καταλήξουμε σε μερικές ενδιαφέρουσες παρατηρήσεις.

- Το CSI snapshot είναι γενικά ταχύτερο σε σχέση με το FSB και το DM, όπως αναμενόταν.
- Το FSB είναι πιο αργό από το CSI αλλά ταχύτερο από το DM. Αυτό έχει λογική, καθώς το DM περιλαμβάνει τη διαδικασία του να πάρει κανείς πρώτα ένα CSI snapshot και στη συνέχεια να δημιουργήσει έναν νέο τόμο από αυτό, εκτελώντας το FSB στον νέο τόμο.
- Το DM είναι πιο αργό από τις δύο αυτές μεθόδους, αλλά παρέχει πιο σταθερό αποτέλεσμα σε σχέση με το FSB, το οποίο εκτελεί την διαδικασία αντιγράφου ασφαλείας στο ζωντανό σύστημα αρχείων.

- Ο μηχανισμός CSI snapshot έχει παρόμοια απόδοση τόσο στην περίπτωση της τροποποίησης (touch) των υπαρχόντων αρχείων όσο και στην προσθήκη νέων (inc) αρχείων.
- Και οι δύο μηχανισμοί που περιλαμβάνουν το FSB (FSB και DM) εκτελούν αξιοσημείωτα καλύτερα με νέα αρχεία σε σχέση με τα τροποποιημένα αρχεία (ακόμα κι αν το μόνο που έχει αλλάξει είναι η χρονική σήμανση του αρχείου).
- Το Velero Plugin για το Longhorn είναι γενικά πιο αργό από το CSI plugin, κάτι που έχει λογική καθώς εκτελεί την ίδια διαδικασία αντιγράφου ασφαλείας και στη συνέχεια συγχρονίζει τα δεδομένα σε ένα επιπλέον bucket.

6 Επίλογος

Πλησιάζουμε στο τέλος αυτής της διπλωματικής εργασίας. Στο κεφάλαιο αυτό, θα προσπαθήσουμε να σκιαγραφήσουμε τη διαδρομή που οδήγησε στην ολοκλήρωσή της και να προτείνουμε μελλοντικά βήματα που θα μπορούσαν να στηριχθούν σε αυτή τη βάση και να την επεκτείνουν.

6.1 Συμπερασματικά Σχόλια

Ξεκινήσαμε αυτή τη διαδικασία εξερευνώντας το Kubernetes και τις δυνατότητές του, βρίσκοντας τρόπους για να αναπτύξουμε και να δοκιμάσουμε clusters τοπικά, και κατανοώντας τις λειτουργίες του Velero. Όταν ήρθε στο προσκήνιο το Longhorn, ο στόχος έγινε πιο ξεκάθαρος: να αναπτύξουμε έναν εύκολο και αυτοματοποιημένο τρόπο για να ενσωματώσουμε τις δυνατότητες του Velero με ένα σύστημα αποθήκευσης που δεν διαθέτει ενσωματωμένη δυνατότητα για αυτό. Ωστόσο, αυτό δεν ήταν αρκετό, καθώς το Longhorn υποστηρίζει το CSI και το Velero μας παρέχει ένα plugin CSI. Στραφήκαμε στην προσθήκη μιας επιπλέον δυνατότητας που θα είχε νόημα για κάθε σύστημα που ακολουθεί την αρχιτεκτονική μας. Εφόσον το Longhorn δεν επιτρέπει τη δημιουργία πολλαπλών Backup Targets, το πρόβλημα έγινε προφανές: η δυνατότητα αποθήκευσης των δεδομένων του αντίγραφου ασφαλείας σε μια δευτερεύουσα τοποθεσία και η αποκατάσταση από αυτήν, εάν χρειαστεί. Από πάνω από όλα αυτά, ο στόχος ήταν ξεκάθαρος: να κάνουμε τα πάντα όσο το δυνατόν πιο αυτοματοποιημένα και να λειτουργούν ομαλά.

Όταν ο στόχος τέθηκε, ο δρόμος περιλάμβανε τη μελέτη του μηχανισμού plugin του

Velero, διάφορων SDK για το Go (συμπεριλαμβανομένων των Longhorn και MinIO), και πολλές απόπειρες για να λύσουμε το παραπάνω πρόβλημα. Η λύση μας περιλάμβανε την συγχρονισμένη αποθήκευση όλων των αντικειμένων κατά τη διάρκεια κάθε αντιγράφου ασφαλείας, έτσι ώστε η δευτερεύουσα τοποθεσία αποθήκευσης να έχει περιεχόμενα ταυτόσημα με την κύρια τοποθεσία που χρησιμοποιήσαμε. Και εάν χρειαστεί να εκτελέσουμε μια διαδικασία αποκατάστασης από τη δευτερεύουσα τοποθεσία αποθήκευσης, η διαδικασία αποκατάστασης απαιτεί από τον χρήστη να ορίσει τη δευτερεύουσα αποθήκευση ως Στόχο Αντίγραφης Ασφαλείας και να εκτελέσει το αντίγραφο ασφαλείας κανονικά.

Τα αποτελέσματα των δοκιμών μας έδειξαν τη σύγκριση μεταξύ των υπαρχουσών λύσεων αντιγράφου ασφαλείας και αποκατάστασης και πώς το Velero plugin για το Longhorn στέκεται απέναντί τους. Επιβεβαίωσαν επίσης το επίπεδο στο οποίο επιτυγχάνει τους στόχους του.

6.2 Μελλοντικό Έργο

Ολοκληρώνοντας αυτή τη διπλωματική εργασία, έγινε σαφές ότι επιτύχαμε τον αρχικό μας στόχο: να κάνουμε τον συγχρονισμό των δεδομένων του τόμου σε μια δευτερεύουσα τοποθεσία όταν χρησιμοποιούμε το Velero με το Longhorn πιο εύκολο. Ωστόσο, θα μπορούσαμε να επεκτείνουμε τη δουλειά που έγινε σε αυτό το έργο με τους εξής τρόπους:

- Να επεκτείνουμε τη λειτουργία συγχρονισμού όχι μόνο για τα δεδομένα του τόμου του αντιγράφου ασφαλείας αλλά και για τα manifests. Το Velero εκτελεί το αντίγραφο ασφαλείας και στη συνέχεια στέλνει τα δεδομένα του αντιγράφου ασφαλείας στην Backup Storage Location που έχει ορίσει ο χρήστης. Θα μπορούσαμε επίσης να συγχρονίσουμε αυτά τα δεδομένα στη δευτερεύουσα τοποθεσία που έχει ορίσει ο χρήστης.

- Να επεκτείνουμε το plugin για να υποστηρίζει ρυθμίσεις που χρησιμοποιούν διαφορετικούς τύπους Backupstores εκτός από το S3, όπως το GCP Cloud Storage, NFS και Azure Blob Storage.

- Να συμβάλουμε και να προωθήσουμε το plugin στο Velero, ώστε το Longhorn να γίνει μέρος των υποστηριζόμενων παρόχων του Velero. [20]

1 Introduction

In this opening chapter, we will outline the course of our work. First, we will discuss the problem and why we are tackling it. Then, we will examine the existing solutions before describing our proposed solution on a high level. Finally, we will demonstrate the structure of this diploma thesis.

1.1 Motivation

Kubernetes has been one of the most significant open-source projects in recent years, which makes it easier for the user to manage containerized applications at scale while also taking advantage of cloud-native architectures. This way, the dynamic management of stateless and stateful applications becomes easier. Various tools have been created to make deploying the Kubernetes cluster easier. One of the most popular is K3S, which is designed to allow the seamless installation of a Kubernetes distribution in resource-constrained environments or even IoT appliances. [1]

Depending on the nature of the application workload, data persistence might be necessary. Consequently, organizations must be able to keep this data safe by deploying effective backup and restore strategies. This way, they can achieve integrity and recoverability, which are crucial principles in data management. In this context, robust backup solutions are critical to avoid data loss and minimize downtime during failures.

A commonly used rule for a successful backup and restore strategy is the ”3-2-1 backup rule”. According to this rule, to protect our data effectively, we need to have three copies of our data stored on two different types of media, with one copy kept off-site. [2] While

this rule dates back to a time when modern technologies (like the cloud) did not exist, it might be wise to use it as a guideline for data redundancy and recovery. Therefore, it is crucial to incorporate the concept of off-site backups. Off-site backup refers to the process of storing data on a remote server or media that is taken to a different location. [3]

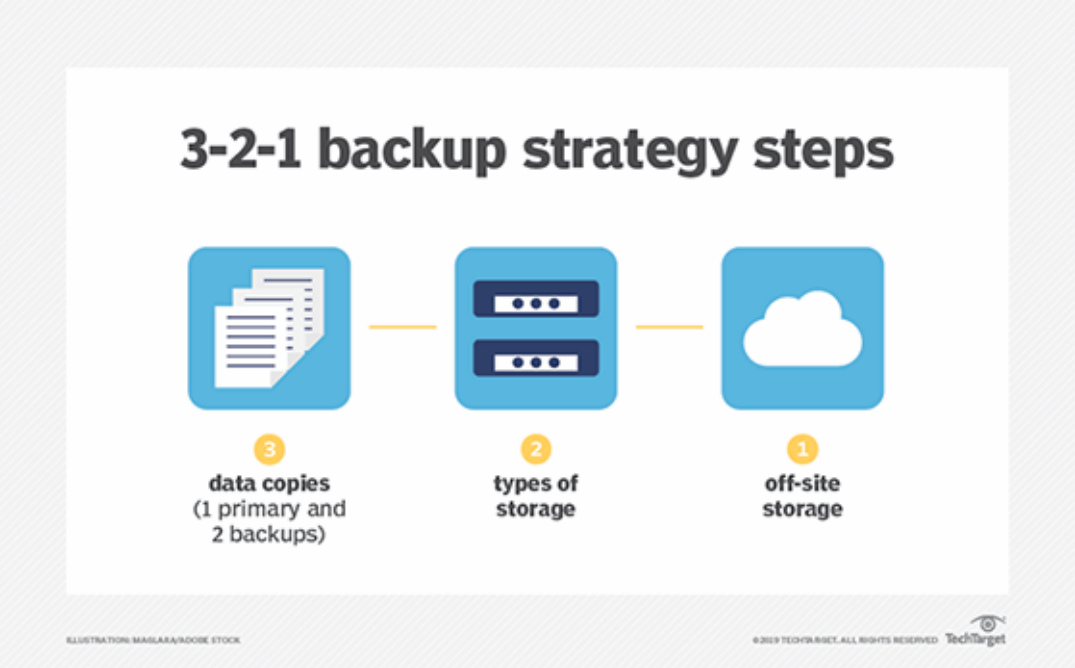


Figure 1.1: *The "3-2-1 backup rule". [21]*

As Kubernetes achieves storage persistence using the concept of Persistent Volumes, the cluster must include a storage solution to manage the creation, the lifecycle, and the deletion of volumes. Longhorn provides these capabilities, along with options to back up and restore these volumes. However, along with the built-in options that Longhorn provides for volume backup, multiple tools have been developed to assist with the process of backing up and restoring a cluster. One of the most popular is Velero, which includes options for backing up and restoring both the Kubernetes resources and volumes containing the applications' data. These tools offer many options for disaster recovery and are designed to provide their capabilities to a wide variety of cluster architectures.

Velero aims to offer a universal solution to the disaster recovery issue for Kubernetes clusters, but this solution cannot directly adapt to the needs of each deployment choice made. Velero has developed a plugin system that helps bridge that gap and offers cluster administrators the chance to have a dedicated plugin that addresses the needs of each

storage system used.

To address the need for off-site backups using Longhorn, we can explore the options that Velero gives us to integrate with other platforms. This way, there will be a direct procedure that will allow effective BR strategies to take place without the user needing to provide additional configuration to the cluster that would otherwise be necessary.

1.2 Problem Statement

As mentioned above, Velero is a tool to safely backup and restore Kubernetes clusters, perform disaster recovery, and migrate Kubernetes cluster resources and persistent volumes. Velero uses plugins for each system to work with the plethora of storage solutions offered in Kubernetes. While there are solutions for Volume Snapshots even in case of storage platforms that do not have a dedicated plugin, they are often limited in capabilities, snapshot consistency or speed.

Additionally, to perform any additional logic on the backup and restore action —like moving the backup to an external cloud storage solution—the user will need to execute manual actions, which increase complexity and the duration of such operations. Due to these reasons, the backup process is not as effective as needed, especially regarding automation and scalability.

To mitigate these shortcomings, a solution is to develop a new plugin with the new functionalities we need. This way, we can introduce a singular solution for the BR strategy using Longhorn and Velero. This strategy will contain all the features we need with as little human intervention as possible.

1.3 Existing Solutions

Velero, one of the most used Kubernetes backup and restore tools, integrates with many storage providers out of the box through its extended plugin system. However, it can also integrate Volume Snapshots with storage systems that do not have a dedicated plugin. There are various ways to achieve this.

- File System Backup.

- CSI Snapshot using a generic CSI plugin.
- CSI Snapshot Data Movement.

Despite these techniques being able to ultimately achieve our goal of successful data protection in Kubernetes clusters, they all come with specific drawbacks. For example, File System Backup backs up data from the live file system, so the entirety of data is not captured at the same point in time. Therefore it is less consistent than the snapshot approaches. [4] In addition, this method utilizes external backup tools like Restic and Kopia, which take snapshots at the file system level. Such backups tend to be significantly slower.

In the case of CSI snapshot, Velero uses a generic CSI plugin. This way can also be used with Longhorn, as Longhorn supports the Kubernetes CSI snapshot mechanism. However, this functionality offers minimal features so that it can be integrated into multiple CSI-compatible systems. The user also needs to enable CSI support on the cluster. CSI support is something that not all Kubernetes distributions include. For example, K3S (the Kubernetes distribution we used in this diploma thesis) does not bundle the Common Snapshot Controller and related Snapshot CRDs. The user needs to employ actions to include these components in the cluster.

Finally, the CSI snapshot data movement feature performs a File System Backup on a volume from which the tool takes a CSI snapshot. This process also demands that the cluster have CSI support enabled. It also makes the snapshot process significantly slower.

Furthermore, all of the above solutions offer a direct backup process, but if we want to apply any custom logic, we must develop our plugin.

1.4 Proposed Solution

Velero's plugin system allows us to develop plugins. These plugins can be of various kinds, including Object Store, Volume Snapshotter, Backup Item Action, Restore Item Action, and Delete Item Action. In our case, we will utilize the VolumeSnapshotter interface. This way, we can set up a plugin that will achieve the goal of off-site backups using Longhorn.

The plugin consists of actions that take place at specific times. First, the user needs to install it at the Velero installation time, when they also need to provide some parameters. Then, when they try to create a new backup, Velero will not only trigger the creation of the Longhorn backup but also wait for it to be created and then sync the Backupstore with an additional external S3 bucket. Velero will trigger the creation of a Longhorn Volume at restore time based on the backup selected. If a disaster occurs on the primary S3 bucket where the Backupstore is hosted, the user can easily set the Backupstore to point to the off-site bucket. Then, the restore process can continue in the same way as before.

This process is as simplified as possible for K3S/Kubernetes clusters with persistent storage. It is also scalable, as even in multiple backups, the user does not need to move the backup data to an external server manually.

As a result, this thesis aims to fill a gap in the domain of Kubernetes backup and restore in clusters that use Longhorn as the storage layer. It attempts to simplify and automate the practice of sending backups to an external site to maintain good disaster recovery practices.

1.5 Outline

The remainder of this diploma thesis is structured in the following way.

- In **Chapter 2**, we provide the necessary background information and context for the reader to follow the subject of this thesis.
- In **Chapter 3**, we explain in detail the design of current solutions when using Velero for Volume Snapshots.
- In **Chapter 4**, we provide a detailed design of the Velero plugin for Longhorn.
- In **Chapter 5**, we evaluate our solutions based on metrics and comment on their effectiveness.
- In **Chapter 6**, we summarize our findings, our contributions, and discuss potential future work.

2 Background

The purpose of this chapter is to explain the technologies used for this thesis and how they integrate. To test the backup and restore strategies for volumes when using Velero, we needed a testing environment that was reproducible, free, and lightweight enough to be set up on a consumer PC. So, we need to describe the features of each one of the tools that we used and how we utilized them to extract the conclusions we finally got. The overall architecture is based on Virtualbox VMs, so we will start from there. Our core tool is Kubernetes. Therefore, we need to explain some basic concepts, while the specific Kubernetes distribution we used is K3S. The block storage layer on which we based our project is Longhorn, and the architecture takes advantage of many of its features. The object storage platform that will be hosting our backup files is MinIO. Finally, the primary tool used to perform all backup and restore operations is Velero, including the backing up and restoring of volumes.

2.1 VirtualBox

2.1.1 Virtual Machines

Since we wanted to simulate a working environment on premises, using a consumer PC, it was necessary to use Virtual Machines that would be used as the hosts of our Kubernetes cluster. A Virtual Machine (VM) is a computing resource that utilizes software rather than a physical machine to execute programs and deploy applications. [5] This is the tool we needed to simulate our hosts. There are two types of Virtual Machines:

a process virtual machine and a system virtual machine. A process VM is a virtual platform created for an individual process and destroyed once the process terminates, while a system VM supports an OS together with many user processes. [6] What we need is a system VM so that we can simulate a full host. The system Virtual Machine concept relies on the existence of a Hypervisor, which is the software that manages (in the sense that it creates, runs, and destroys) all VM's on the system. There are two types of Hypervisors:

- Type 1 (or "bare metal") hypervisor. This type is essentially a lightweight operating system which runs directly on the system's hardware.
- Type 2 (or "hosted") hypervisor. This type runs as a normal application on top of the operating system and therefore, all system calls to the system's hardware pass through the operating system.

2.1.2 Why use VirtualBox

VirtualBox is a type 2 hypervisor. It functions as an app that runs on top of the existing Operating System. Windows has Hyper-V hypervisor activated by default (Hyper-V is a type 1 hypervisor), which interacts directly with the hardware prior to any operating system loading, thus causing conflicts with other applications that require "virtualization technology," such as VirtualBox, which is classified as a type-2 hypervisor. [22] Despite VirtualBox being able to run at the same time as Hyper-V (note that this is an experimental feature), it is recommended to turn off Hyper-V, as enabling this feature may lead to noticeable performance issues with Oracle VM VirtualBox on certain host systems. [23] To do this we need to run the following command in Windows CMD (run in Administrator Mode).

```
bcdedit /set hypervisorlaunchtype off
```

This way, we can fully utilize VirtualBox's features, without any performance issues.

2.1.3 Specifications of VM's used

As mentioned earlier, VirtualBox is the platform we are using to create the VM's that function as the nodes of the K3S cluster. The characteristics of each VM used as a K3S

node can be seen in Table 2.1. Theses specifications are discussed more in the K3S section.

Base Memory	8192 MB
Processors	2
Virtual Hard Disk	200 GB
Network Adapters	Bridged Network

Table 2.1: *The characteristics of the VM's used for K3S nodes.*

Regarding networking, our VM's have 1 network card each. The network card has the `Bridged Network` setting. With bridged networking, VirtualBox uses a device driver existing on the host system, which filters data from the physical network adapter. This driver is called a `net filter` driver. Once data comes through this driver, VirtualBox intercepts it and inserts data, so that the host system interprets the packet received as if it comes from a directly connected network interface. This essentially creates a new "software network card".

The Operating System used is `Ubuntu Server 23.10 (Mantic Minotaur)` with Linux Kernel version `6.5.0-44-generic`.

In the case of the VM used to host the MinIO Object Storage, the specification differ a little. This is due to the different requirements that the software hosted on each VM have. The specifications of this VM can be seen in Table 2.2.

Base Memory	4096 MB
Processors	2
Virtual Hard Disk	100 GB
Network Adapters	Bridged Network

Table 2.2: *The characteristics of the VM's used for MinIO nodes.*

The Operating System and kernel used for the MinIO host is the same as for the K3S hosts: Ubuntu Server 23.10 (Mantic Minotaur) with Linux Kernel version of 6.5.0-44-generic.

2.2 Kubernetes

2.2.1 VM's vs Containers

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. link Kubernetes at its core, makes use of containers, which are an alternative means of virtualization compared to VM's.

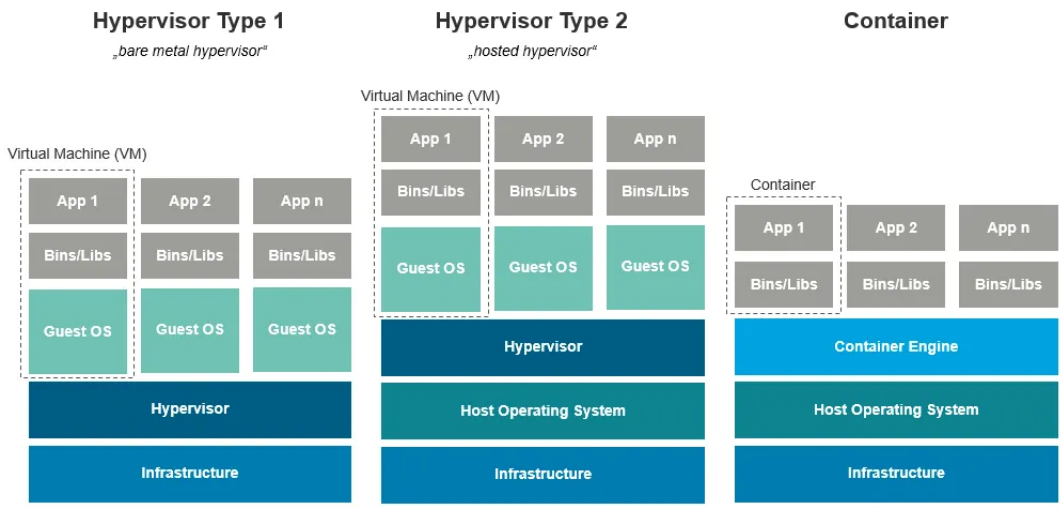


Figure 2.1: *VM vs Container comparison. [24]*

As mentioned above, VM's use software to emulate a hardware system. They contain the whole operating system, along with the binaries, the libraries and the applications that it packages. They also demand the existence of a hypervisor, which is the software that manages all VM's. On the contrary, a container essentially shares the operating system of its host and it only packages all the components needed in the application layer, including binaries, libraries and the code of the application. Therefore, it is more lightweight compared to a VM. In addition, containerized software offers multiple advantages involving the ease of image creation, portability regarding the OS that it can be run on (including cloud and on-prem) solutions and better resource isolation and utilization. [10]

2.2.2 Kubernetes basics

Kubernetes is a Container Orchestration (CO) system. These systems can be used to automatically provision, deploy, scale, and manage containerized applications anywhere with minimal interference with the underlying infrastructure. [7] As containerized applications grew in complexity and quantity, the need for a central software tool that is able to orchestrate the workloads, became clear. Kubernetes is a tool that can address this need, as it schedules and automates container-related tasks from the initiation of the application up to its end. [8] These tasks, include the following among more.

- Deployment: Deploy a specified number of containers to a specified host and keep them running in a wanted state.
- Rollouts: A rollout is a change to a deployment. Kubernetes lets you initiate, pause, resume or roll back rollouts.
- Service discovery. Kubernetes can automatically expose a container to the internet or to other containers by using a domain name system (DNS) name or IP address.
- Storage provisioning: Set Kubernetes to mount persistent local or cloud storage for your containers as needed.
- Load balancing: Based on CPU usage or custom metrics, Kubernetes load balancing can distribute the workload across the network to maintain performance and stability.
- Autoscaling: When traffic spikes, Kubernetes autoscaling can spin up new clusters as needed to handle the additional workload.
- Self-healing for high availability: When a container fails, Kubernetes can restart or replace it automatically to prevent downtime. It can also take down containers that don't meet your health check requirements.

Kubernetes can be deployed either on premises or at a cloud platform. Many cloud platforms also offer managed Kubernetes solutions, which take care of various tasks. These tasks can bring a significant overhead to the teams that manage them, and they

can include detached credential configuration, self-recovery, batch execution, workload management, progressive application deployment, and more. [9]

2.2.3 Kubernetes architecture

Deploying Kubernetes, means creating a `Cluster` of nodes. A Kubernetes cluster consists of a set of worker machines, called `Nodes`, that run containerized applications. Every cluster has at least one worker node. [10]

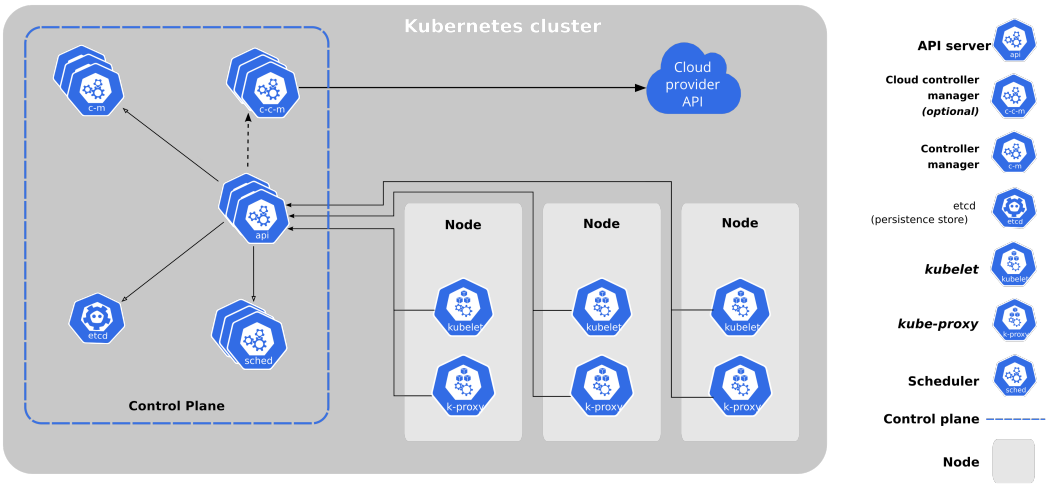


Figure 2.2: *The basic components of Kubernetes. [25]*

The worker node(s) host the `Pods` that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. Below, we describe some of the main Kubernetes components.

- kube-apiserver: the API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver.
- etcd: it is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
- kube-scheduler: it is a control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

- kube-controller-manager: it is a control plane component that runs controller processes.

- cloud-controller-manager: it is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets us link our cluster into our cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with our cluster.

2.2.4 Storage in Kubernetes

Kubernetes can be used with no problem to run stateless applications, which do not require the existence of persistence data. However, in most scenarios the applications are stateful, and a kind of persistence of data is required to execute complex scenarios, so that the data stored in a cluster can be accessed even when a pod is deleted or restarted. In that case, Kubernetes has various components that allow us to handle storage.

Volumes

Kubernetes deletes all data saved on a container when it is crashed or stopped. However, it does not delete data on a Volume and the Volume's data is persisted across pod restarts. A Volume is essentially a directory, possibly with some data in it, which is accessible to the containers in a pod. All the underlying details regarding the details of this directory, are dependent on the type of Volume used. [26]

Persistent Volumes

The way that Kubernetes users and administrators can interact with storage in Kubernetes includes the introduction of two resources: the PersistentVolume (PV) and the PersistentVolumeClaim (PVC). A PersistentVolume is a resource that is cluster-wide (that means, it is not namespaced) and represents a piece of storage. It can be provisioned either statically by an administrator, or dynamically. It is not dependent on the lifecycle of the pod that it has been mounted on, that means that even when the pod crashed or is

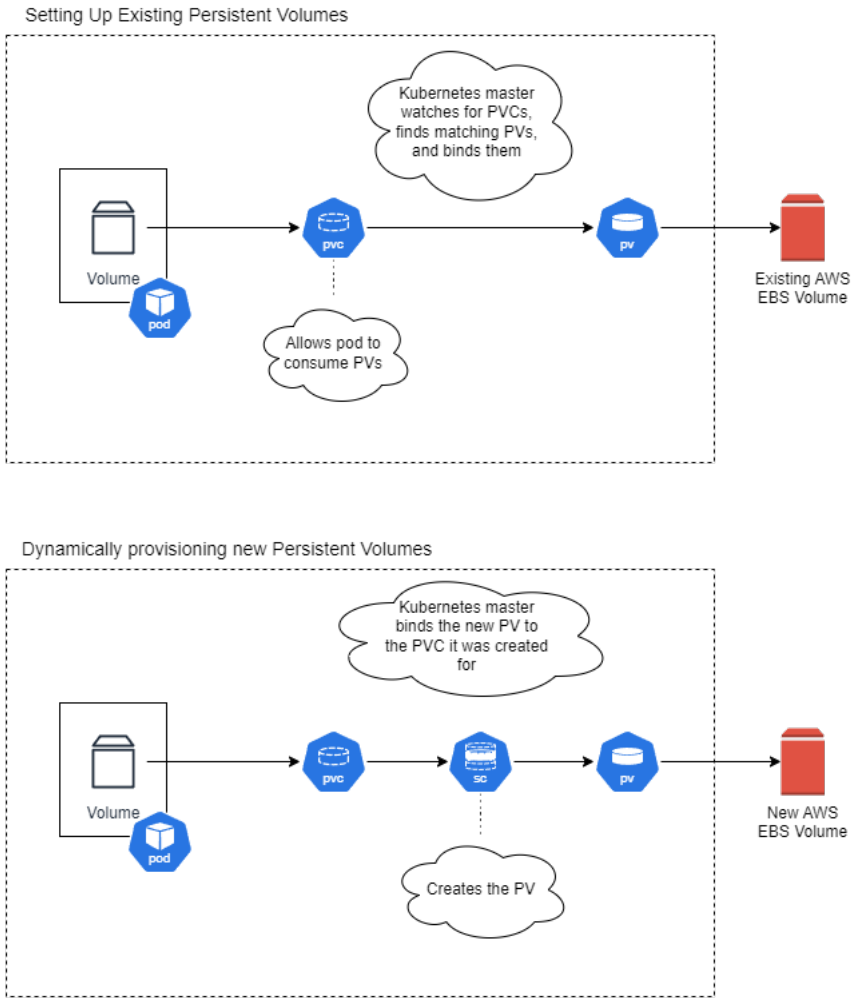


Figure 2.3: *Static vs Dynamic provisioning of a PersistentVolume.* [27]

stopped, the PersistentVolume resource persists. The type of the PersistentVolume defines its properties. The types of PersistentVolumes include csi, fc, hostPath, iscsi, local and nfs.

Persistent Volume Claims

A PersistentVolumeClaim is a request for storage by a user. It is a namespaced resource, and it consumes resources of a PersistentVolume. The relationship between PV and PVC is one-to-one.

Static vs Dynamic Provisioning

When we need to attach a Volume to a pod, we can create the Volume either statically or dynamically. When using static provisioning, the cluster administrator would normally have to manually create new storage volumes through the cloud or storage provider, and then create PersistentVolume objects to represent these in Kubernetes. However, using the dynamic provisioning feature, the volume can be created dynamically after the creating of the PersistentVolumeClaim. The Kubernetes object responsible for this is the Storage Class.

Storage Class

A StorageClass provides a way for administrators to describe the classes of storage they offer. Each StorageClass in a cluster represents a different ”type” of storage and the cluster can contain multiple storage classes. Each StorageClass contains the fields *provisioner* that determines what volume plugin is used for provisioning PVs, *parameters* for any storage specific parameters and *reclaimPolicy* which determines what happens to a PV provisioned by the StorageClass, after the corresponding PVC is deleted. When a user creates a PersistentVolumeClaim, they specify a name for the StorageClass that they want to provide the new Volume. In case they don't set any name for the StorageClass, there is a default StorageClass that will be used.

Container Storage Interface

Container Storage Interface (CSI) is a standard for exposing arbitrary block and file storage systems to containerized workloads on Container Orchestration Systems (COs) like Kubernetes. [28] CSI was implemented as a means to remove ”in-tree” volume plugins which were difficult to add and maintain. [29] With the introduction of CSI Kubernetes became extensible on the field of storage, as third-party storage providers can write and deploy plugins exposing new storage systems in Kubernetes without ever having to touch the core Kubernetes code.

Any vendor that wishes to develop a CSI driver needs to include some CSI sidecar containers, which are developed by the Kubernetes Storage community. [30] These containers include:

- external-provisioner: watches Kubernetes VolumeAttachment objects and triggers ControllerPublish and ControllerUnpublish operations against a CSI endpoint.
- external-attacher: watches Kubernetes PersistentVolumeClaim objects and triggers CreateVolume and DeleteVolume operations against a CSI endpoint.
- external-snapshotter: watches Kubernetes VolumeSnapshot CRD objects and triggers CreateSnapshot and DeleteSnapshot operations against a CSI endpoint.
- external-resizer: watches the Kubernetes API server for PersistentVolumeClaim object edits and triggers ControllerExpandVolume operations against a CSI endpoint if user requested more storage on PersistentVolumeClaim object.
- node-driver-registrar: registers the CSI driver with kubelet using the kubelet device plugin mechanism.
- cluster-driver-registrar (deprecated): registers a CSI Driver with the Kubernetes cluster by creating a CSIDriver object which enables the driver to customize how Kubernetes interacts with it. Since this container has been deprecated, developers and CSI driver vendors will now have to add a CSIDriver object in their installation manifest or any tool that installs their CSI driver. [31]
- livenessprobe: monitors the health of the CSI driver and reports it to Kubernetes via the Liveness Probe mechanism.

CSI Snapshot CRDs

For CSI to function, we need 3 new Custom Resource Definitions (CRDs). These are the VolumeSnapshot, the VolumeSnapshotContent and the VolumeSnapshotClass CRDs. In Kubernetes, a VolumeSnapshot represents a snapshot of a volume on a storage system. [33] As shown in Figure 2.5, the VolumeSnapshot is related to the VolumeSnapshotContent in the same way that a PersistentVolumeClaim is related to a PersistentVolume. A VolumeSnapshot is a request by a user for the storage system to take a snapshot of a volume. A VolumeSnapshotContent is the resource that represents the snapshot in the cluster, and it can either be pre-provisioned by the administrator of the cluster or

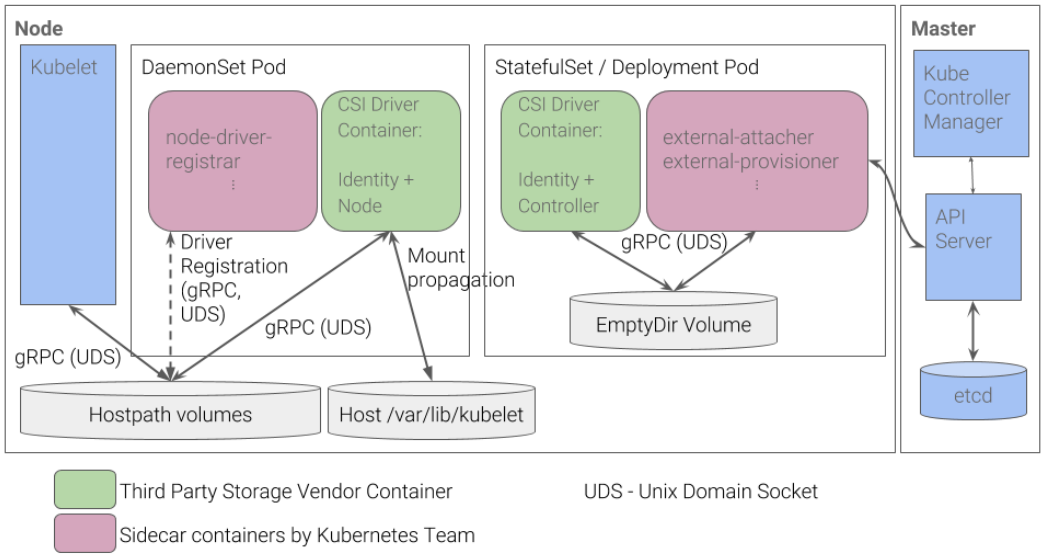


Figure 2.4: *Recommended Mechanism for Deploying CSI Drivers on Kubernetes. [32]*

dynamically provisioned. If we want the VolumeSnapshotContents to be dynamically provisioned, we can use the VolumeSnapshotClass. VolumeSnapshot is only available for CSI drivers in Kubernetes and since we are using Longhorn, we can utilize this feature.

> VolumeSnapshot, VolumeSnapshotContent and VolumeSnapshotClass are all CRDs, not part of the core API.

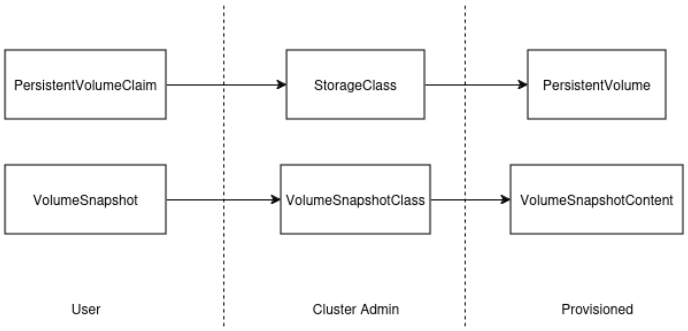


Figure 2.5: *VolumeSnapshot and VolumeSnapshotContent.*

A VolumeSnapshotClass is a way to describe the "classes" of storage when provisioning a volume snapshot. [34] Each VolumeSnapshotClass contains the fields *driver*, *deletionPolicy*, and *parameters*, which are used when a VolumeSnapshot belonging to the class needs to be dynamically provisioned. *driver* is a field that must be specified, as it deter-

mines what CSI volume plugin is used for provisioning *VolumeSnapshots*. Depending on the driver, different parameters may be accepted.

2.3 K3S

K3S is a lightweight Kubernetes distribution, that can be installed in a wide variety of environments. It was selected, as it offers multiple advantages such as being really lightweight (which is important in a home environment), ease of installation and use, and support of Longhorn (our storage solution of choice). An additional advantage of K3S is that it bundles required dependencies for multiple helpful features: [11]

- containerd / cri-dockerd container runtime (CRI)
- Flannel Container Network Interface (CNI)
- CoreDNS Cluster DNS
- Traefik Ingress controller
- ServiceLB Load-Balancer controller
- Kube-router Network Policy controller
- Local-path-provisioner Persistent Volume controller
- Spegel distributed container image registry mirror
- Host utilities (iptables, socat, etc)

2.3.1 K3S Architecture

K3S comprises of two types of nodes: [35]

- A server node runs the control-plane (Kubernetes API, controller, and scheduler), SQLite as the default storage backend and a reverse tunnel proxy, which eliminates the need for bidirectional communication between server and agent.

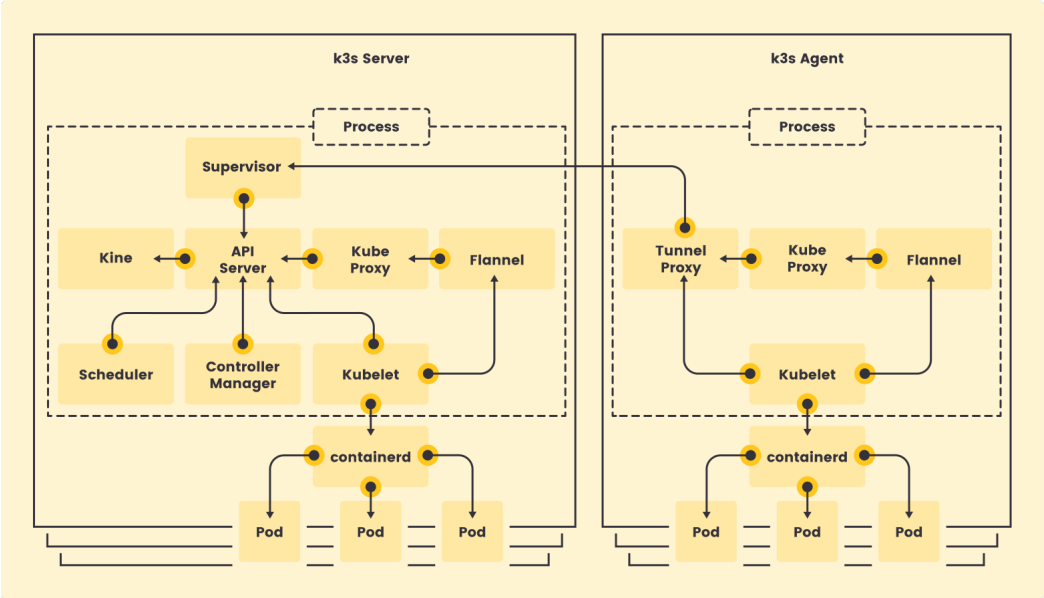


Figure 2.6: *K3S Architecture. [37]*

- An agent node runs the kubelet and kube-proxy. Additionally, it runs Flannel as embedded process, containerd as the container runtime, internal load balancer that load-balances connections between all API servers in HA configurations and network policy controller to enforce network policies.

Both of these node types run the kubelet, container runtime and CNI. K3S also replaces etcd with Kine (*Kine is not etcd*). Kine is a compatibility layer to etcd developed by the K3S team that translates etcd API to SQLite, Postgres, MySQL/MariaDB or NATS. [36] It accepts etcd v3 requests from Kubernetes, translates these to SQL queries, and sends them to a database backend. [35] K3S specifically uses by default the SQLite variant.

2.3.2 Volumes and Storage in K3S

If we need to deploy a stateful app in Kubernetes, we must find a solution for persistent storage. In K3S, we have two options for data persistence: either the Local Storage Provider or Longhorn.

K3S includes Rancher's Local Path Provisioner, allowing the creation of persistent volume claims using local storage on each node by default. [38] The user can configure the provisioner to create either *local* or *hostPath* volumes. As the Kubernetes Local Volume provisioner cannot dynamically provision local volumes, Rancher's solution comes to

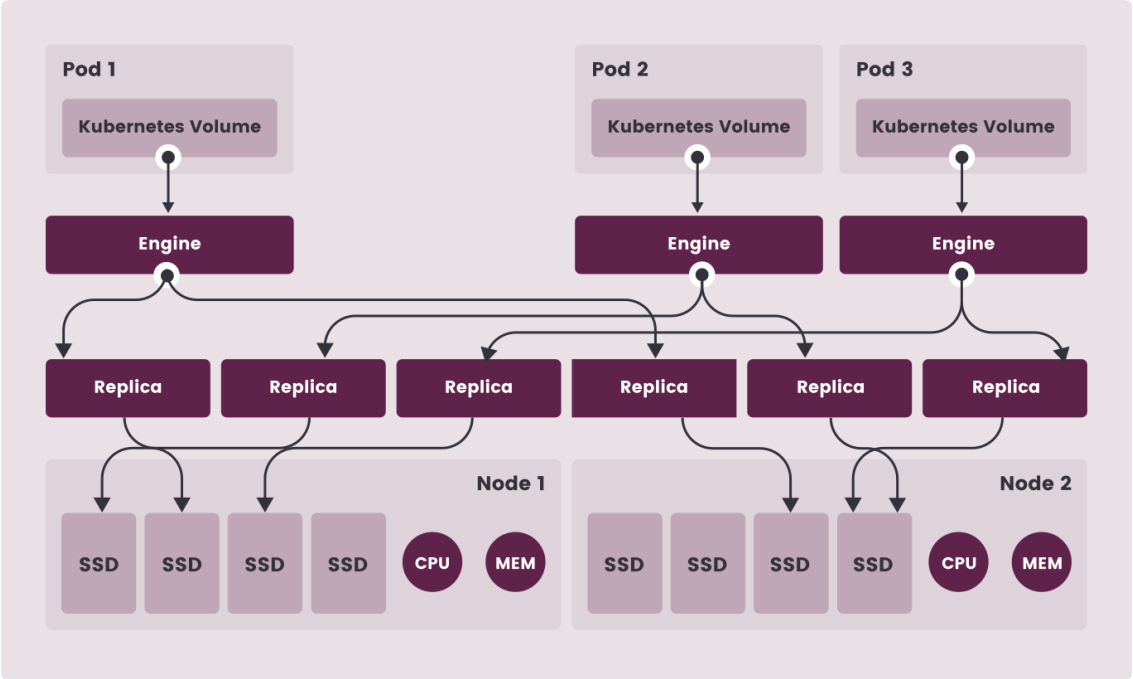


Figure 2.7: *Read/write Data Flow between the Volume, Longhorn Engine, Replica Instances, and Disks. [13]*

solve this issue. [39]

K3S also supports Longhorn as storage backend, which will be examined in detail next.

2.4 Longhorn

Longhorn is a cloud native block storage solution for Kubernetes. Longhorn implements distributed block storage using containers and microservices. It creates a dedicated storage controller for each block device volume and synchronously replicates the volume across multiple replicas stored on multiple nodes. These replicas consist of a chain of snapshots, showing a history of the changes in the data within a volume. The storage controller and replicas are themselves orchestrated using Kubernetes. [12] This simplifies the complexity of distributed storage, by essentially making each volume its own microservice. The controller of each volume is called the *Longhorn Engine* and the component that orchestrates all of these together is called the *Longhorn Manager*.

2.4.1 Longhorn Manager

The Longhorn Manager is a Daemonset and it runs one pod on each node. It is responsible for creating and managing volumes within the Kubernetes cluster and processes API calls from the UI or volume plugins. [13] Longhorn Manager follows the Kubernetes controller pattern and is responsible for replicating and managing Longhorn Volumes. When the Longhorn manager detects that it must create a new node, it also creates an instance of the Longhorn Engine on the node where the volume is attached to and replicas of the volume, where each replica consists of a series of snapshots of the volume. [40] These replicas should always be placed on separate nodes to ensure maximum availability.

2.4.2 Longhorn Engine

The Longhorn Engine is a lightweight block device storage controller, which is capable of storing data of a volume in multiple replicas. [41] It always runs in the same node as the Pod that uses the Longhorn volume and synchronously replicates the volume across the multiple replicas stored on multiple nodes.

2.4.3 Longhorn and CSI

Longhorn provides us with a Container Storage Interface (CSI) driver (*driver.longhorn.io*). This driver takes the block device, formats it, and mounts it on the node. Then the kubelet bind-mounts the device inside a Kubernetes Pod. This allows the Pod to access the Longhorn volume. [13] Longhorn can be managed in Kubernetes by a CSI plugin. This plugin can execute calls to the Longhorn to create, delete, attach, detach, mount a volume, and take snapshots of a volume. The Kubernetes cluster internally communicates with the Longhorn CSI plugin using the CSI interface, while the plugin communicates with the Longhorn Manager using the Longhorn API, as depicted in Figure 2.8.

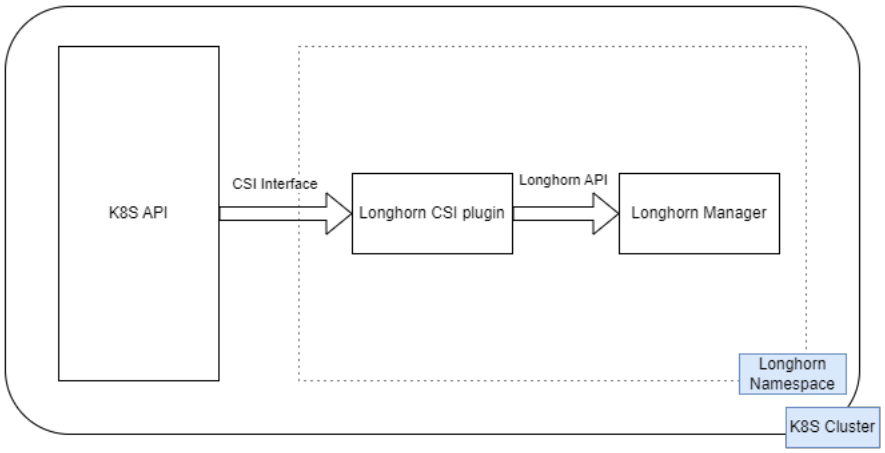


Figure 2.8: *Calls between Kubernetes API, Longhorn CSI plugin and Longhorn Manager.*

2.4.4 Longhorn Volumes Creation

When creating a volume, the Longhorn Manager creates the Longhorn Engine microservice and the replicas for each volume as microservices. Together, these microservices form a Longhorn volume. Each replica should be placed on a different node or on different disks. After the Longhorn Engine is created by the Longhorn Manager, it connects to the replicas. The Engine exposes a block device on the same node where the Pod is running.

2.4.5 Longhorn Volumes naming

Longhorn Volumes can be either statically or dynamically created. [42]

- When created statically (through Longhorn UI), the user can define a name of their choice. To be able to attach the volume to a Kubernetes pod and use it, the user also needs to create a PV for the created Longhorn Volume and also create a PVC that requests storage from this PV. The manifests for these resources can be like the following:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: my-longhorn-pv
spec:
capacity:
    storage: 10Gi  # Existing Longhorn Volume size
accessModes:
    - ReadWriteOnce
persistentVolumeReclaimPolicy: Retain
storageClassName: longhorn  # Storage Class used
csi:
```

```
    driver: driver.longhorn.io
    volumeHandle: <volume-name>  # Existing Longhorn Volume
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-longhorn-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi  # Capacity of the PV
  storageClassName: longhorn  # Storage Class used
```

- When the Volume is provisioned dynamically with the creation of a PVC, the CSI provisioner will generate a name for the provisioned PV. This name will consist of a standard prefix (which has the value ”pvc” by default) and a UUID. The user can alter this prefix by adding the `–volume-name-prefix` command line argument during the deployment of the provisioner. The length of the prefix can also be altered using the `–volume-name-uuid-length` command line argument. The default behavior is to not truncate the UUID (128-bits or 36 characters including 4 dashes). The Longhorn Manager also uses the name that the CSI provisioner generates and assigns to the PV for the creation of the Longhorn Volume. [43]

2.4.6 The Longhorn UI

Longhorn provides the user with a UI, so that they can perform all necessary actions. These actions include managing volumes, snapshots, backups, nodes and disks. Additionally, the user can monitor the state of Longhorn and get information regarding all Longhorn resources. The UI interacts with the Longhorn Manager through the Longhorn API, and acts as a complement of Kubernetes.

2.5 MinIO

MinIO is a high-performance object storage system. It is Kubernetes-native and can be used to provide data management solutions in applications utilizing Kubernetes. It is fully compatible with Amazon S3, therefore it can serve as a free and open source alternative for development and testing environments. It comes in various versions, including deployment options for Kubernetes, Docker, Linux, MacOS and Windows. MinIO can be deployed either in a standalone or a distributed configuration. [14]

MinIO offers multiple features, some of which are erasure coding, bitrot protection and encryption.

Erasure Coding

Erasure Coding is a feature that provides data redundancy and availability. MinIO uses Reed-Solomon code to stripe objects into data and parity blocks, and the user can configure the redundancy levels based on their use case.

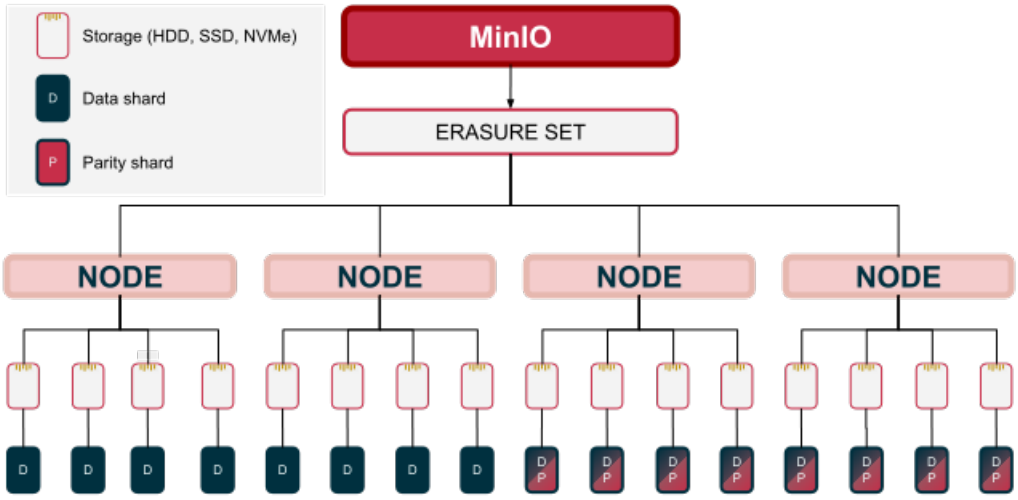


Figure 2.9: *Erasure coding example with an erasure set of 16 drives. This can support parity between EC:0 and 1/2 the erasure set drives, or EC:8. [44]*

The maximum parity that the user can set is N/2, where N equals the number of the available drives in our setup. Using Erasure Coding, MinIO can ensure uninterrupted read and write operations with only ((N/2)+1) operational drives in the deployment.

When writing data, MinIO partitions the object into `data` and `parity` shards. The parity that the user can set, determines how many parity shards MinIO will generate. The formula on which this process is based is the following:

$$\text{N (ERASURE SET SIZE)} = \text{K (DATA)} + \text{M (PARITY)}$$

The maximum parity value is N/2 (or 1/2 of the Erasure Set size).

Bitrot Protection

In case the data in a drive is corrupted for any reason (for example aging drives, current spikes or driver errors), MinIO provides with the Bitrot Protection feature, which ensures that it doesn't read corrupted data from the drives. This is achieved using a custom implementation of the HighwayHash algorithm, an algorithm introduced by Google that provides us with low latency even for small inputs. [45] MinIO computes a hash on a read operation and verifies it on a write operation of the application, across the network and to the memory/drive. The primary focus of this implementation is speed and can achieve hashing speeds over 10 GB/sec on a single core on Intel CPUs. [46]

Encryption

MinIO uses encryption on two levels: when it is transmitted over the network and when it is stored on drives. [47]

In the first case, MinIO ensures the confidentiality of data when it is sent between external applications and MinIO, as well as between nodes within the MinIO cluster. For this cause, it supports Transport Layer Security (TLS) v1.2+ to encrypt all network traffic.

In the second case, MinIO uses authenticated encryption with associated data (AEAD) to maintain the confidentiality and authenticity of data. AEAD encrypts and authenticates plain text data to produce ciphertext and an authentication code. If an unauthorized access were to corrupt the data, even something as small as changing a single bit, the decryption and verification routine would detect the modification using the authentication code.

MinIO AEAD encryption supports protocols such as AES-256-GCM and ChaCha20-Poly1305 to secure object data and users can enable automatic bucket-level encryption to en/decrypt objects as they are written to or read from object storage.

MinIO Architecture

MinIO can be deployed in two ways: Standalone and Distributed. Standalone MinIO can be installed on a single server with either one drive (Single-Node Single-Drive - SNSD) or multiple drives (Single-Node Multi-Drive - SNMD). SNSD configuration

is only suitable for early development and evaluation, as it uses a zero-parity erasure coded backend that provides no added reliability or availability beyond what the underlying storage volume implements. SNMD deployments provide drive-level reliability and failover/recovery with performance and scaling limitations imposed by the single node. Finally, the distributed topology (Multi-Node Multi-Drive - MNMD) provides a enterprise-grade performance, availability, and scalability and are the recommended topology for all production workloads. For a distributed deployment, at least 4 MinIO hosts with homogeneous storage and compute resources are necessary. [48]

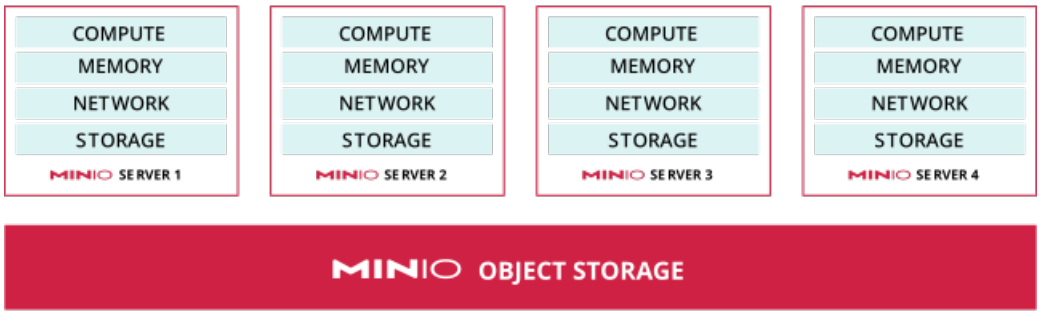


Figure 2.10: *MinIO distributed storage.*

2.6 Velero

Velero is an open source tool that allows users to back up and restore their Kubernetes cluster resources and persistent volumes. [15] It consists of a server that runs on the cluster and a command-line client that runs locally.

Backup Sequence

Velero is based on Custom Resource Definitions (CRDs) for its operations (Backup, Restore etc). It follows the controller pattern, as its modules contain controllers which monitor the state of the resources and then act. [49] In Figure 2.11 we can see how this works when creating a backup.

1. [**Velero client**] Make a call to the Kubernetes API server to create a Backup object.
2. [**BackupController**] Notice the new Backup object and performs validation.

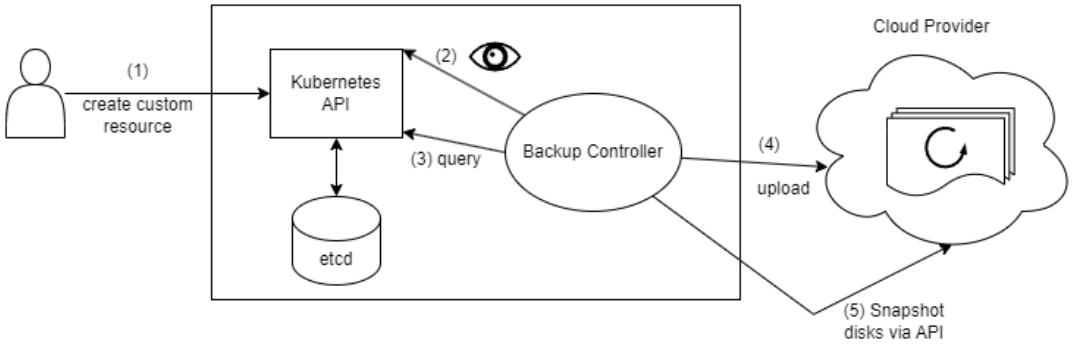


Figure 2.11: *Velero backup process. [50]*

3. [**BackupController**] Begin the backup process. It collects the data to back up by querying the API server for resources.

4. [**BackupController**] Make a call to the object storage service – for example, AWS S3 – to upload the backup file.

Restore Sequence

In Figure 2.12, we can see the restore sequence.

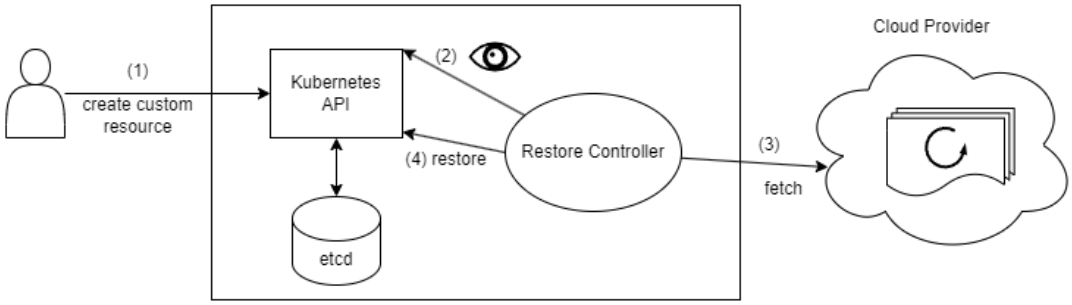


Figure 2.12: *Velero restore process. [50]*

1. [**Velero client**] Make a call to the Kubernetes API server to create a Restore object.

2. [**RestoreController**] Notice the new Restore object and performs validation.

3. [**RestoreController**] Fetch the backup information from the object storage service. It then runs some preprocessing on the backed up resources to make sure

the resources will work on the new cluster. For example, using the backed-up API versions to verify that the restore resource will work on the target cluster.

4. [**RestoreController**] Start the restore process, restoring each eligible resource one at a time.

Velero Plugins Mechanism

Velero uses a plugin system to interact with multiple storage systems. Users can add their own plugins to implement their own custom functionality to Velero backups & restores without having to modify/recompile the core Velero binary. Users can create their image which contains any extra functionalities and this image is deployed as an init container for the Velero server pod and copies the binary into a shared emptyDir volume for the Velero server to access.

Plugins can be of one of the following types:

- Object Store: Persists and retrieves backups, backup logs and restore logs.
- Volume Snapshotter: Creates volume snapshots (during backup) and restores volumes from snapshots (during restore).
- Backup Item Action: Executes arbitrary logic for individual items prior to storing them in a backup file.
- Restore Item Action: Executes arbitrary logic for individual items prior to restoring them into a cluster.
- Delete Item Action: Executes arbitrary logic based on individual items within a backup prior to deleting the backup.

A plugin can serve more than one of these roles at the same time. It is necessary to specify at least one plugin at installation time, which needs to be of Object Store or VolumeSnapshotter type or a plugin that contains both. This can be specified with the use of the `-plugins` flag. When no Backup Storage Location or Volume Snapshot Location is specified, this flag is optional.

Velero offers plugins for the following storage systems.

Storage Provider	Type	Notes
Amazon Web Services (AWS)	Object Storage VolumeSnapshotter	Supported by Velero team
Google Cloud Platform (GCP)	Object Storage VolumeSnapshotter	Supported by Velero team
Microsoft Azure	Object Storage VolumeSnapshotter	Supported by Velero team
VMware vSphere	VolumeSnapshotter	Supported by Velero team
Alibaba Cloud	Object Storage VolumeSnapshotter	
DigitalOcean	Object Storage VolumeSnapshotter	
HPE Storage	VolumeSnapshotter	
openEBS	VolumeSnapshotter	
OpenShift	BackupItemAction RestoreItemAction	
Portworx	VolumeSnapshotter	
Storj	ObjectStorage	
Container Storage Interface (CSI)	VolumeSnapshotter	Supported by Velero team
OpenStack	Object Storage VolumeSnapshotter	

2.7 System Architecture

The above tools were all used to set up a testing workspace. The complete architecture of the system and how they integrate with each other, can be seen in the diagram below. First of all, the system is set up on a personal on premises computer (the ”Host Computer”). This computer has VirtualBox installed, and three VirtualBox VMs have been deployed: two for the K3S nodes and one for MinIO.

- VirtualBox VM #1: It is used as the host for the Master K3S node (or as called in K3S, the ”Server Node”). This means that it includes all the resources necessary for the Control Plane of Kubernetes, the dedicated Longhorn resources and the resources needed for Velero.
- VirtualBox VM #2: It hosts any resources responsible for the participation in the cluster (e.g. kubelet, kube-proxy) and all Longhorn and Velero resources.
- VirtualBox VM #3: It hosts the Standalone MinIO Server.

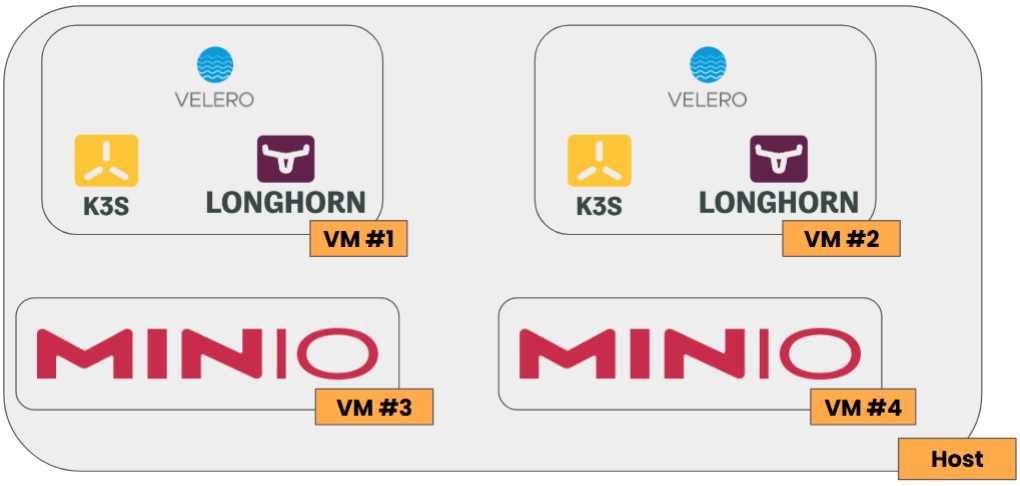


Figure 2.13: *The architecture of the system.*

3 Design

The process of creating a backup in a Kubernetes environment which utilizes Longhorn volumes using Velero can take multiple forms. In this chapter, we are going to examine the various available options, based on the defining characteristics of each.

3.1 Different Backup Types

Velero lets us backup clusters, including any volumes attached to the pods. When using the above architecture, there are three mechanisms that Velero utilizes to maintain the state of volumes.

- CSI Snapshot
- File System Backup
- CSI Snapshot Data Movement

In the following subsections, we will examine how each method works internally and how we can implement it and incorporate it into our cluster.

3.1.1 CSI Snapshot

CSI Snapshot utilizes the support of CSI (Container Storage Interface) in Velero. As mentioned in Chapter 2, Longhorn also utilizes CSI. Longhorn's CSI plugin allows us to

take snapshots of our volumes, and these snapshots can be either Longhorn Snapshots, Longhorn Backups and Longhorn Backing Images.

How Longhorn Snapshots work

A Longhorn Snapshot is a point-in-time copy of a volume in the Longhorn distributed block storage system. Longhorn Snapshots are incremental, which means that after the initial snapshot that the user creates, all subsequent snapshots only record the changes -e.g. the different blocks, since Longhorn operates on block level- made to the volume since the last snapshot, thus creating a chain of snapshots.

Snapshots are stored locally in each node, as a part of each replica of a volume, as replicas are essentially a chain of Longhorn Snapshots. They are stored on the disk of the nodes within the Kubernetes cluster in the same location as the volume data on the hosts physical disk. The path where the snapshot data is stored in each node is `/var/lib/longhorn/replicas`. [51]

Each Longhorn Snapshot is represented internally by two files. For example, for a volume called *pvc-<pvc_uuid>* and the snapshot called *snapshot-<snapshot_uuid>* we will have two files:

- *volume-snap-snapshot-<snapshot_uuid>.img*
- *volume-snap-snapshot-<snapshot_uuid>.img.meta*

The *.img* file contains the volume data, while the *.img.meta* file is a JSON file which contains the metadata of the snapshot.

```
{
    "Name": "<Name of the Snapshot>",
    "Parent": "<Parent Snapshot>",
    "Removed": "<Marked for Removal - true or false>",
    "UserCreated": "<Created by the User - true or false>",
    "Created": "<Time and Date>",
    "Labels": "null"
}
```

Longhorn Snapshots are involved when reading from a Longhorn Volume. When data is read from a replica of a volume, Longhorn first checks the live data. If the data isn't there, it searches through snapshots, starting with the latest and moving to older ones

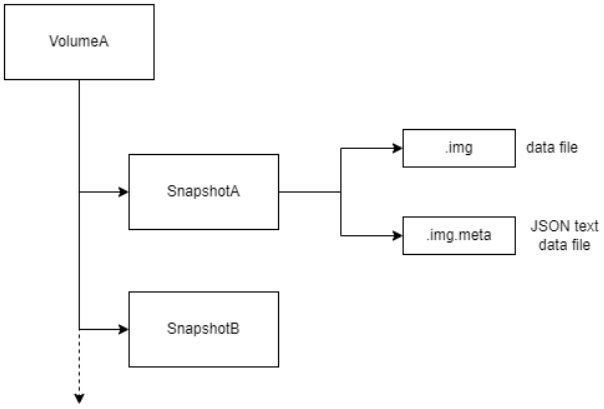


Figure 3.1: *The contents of a Longhorn Snapshot*

until the data is found. Since snapshots are incremental, the differencing disk chain (also called a chain of snapshots) can get quite long. To improve read performance, Longhorn maintains a read index that records which differencing disk holds valid data for each **4K block of storage**. [52]

Longhorn Snapshots are immutable. This means that the user cannot modify the files of this snapshot after their creation. If additional changes need to be captured, a new snapshot must be made.

The only case in which the files of an existing snapshot can be changed, is if the previous snapshot is deleted by the user. In this case, Longhorn merges the changes it holds with the next most recent one in the snapshot files. The user won't notice any difference in the remaining snapshots, except for a possible change in the size of the next snapshot, depending on the deleted one's contents.

Any new data that the user writes to the volume, is written to the live version, which acts as a snapshot of the latest changes compared to the previous snapshot.

To make the Longhorn Snapshots function clearer, consider the following example:

The smallest Longhorn Volume that the user can create is of size 10MB. To examine this condition we can create a pod using a Longhorn Volume of this size and its corresponding PVC.

```
#small-volume-example.yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  namespace: default
spec:
  containers:
```

```
    - name: my-container
      image: ubuntu
      command: ["/bin/bash", "-c", "sleep 3600"]
      securityContext:
        privileged: true
      volumeMounts:
        - name: test-storage
          mountPath: /mnt/longhorn
  volumes:
    - name: test-storage
      persistentVolumeClaim:
        claimName: test-pvc
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Mi
  storageClassName: longhorn-new
```

Notice that the pod need to run in privileged mode, so that we can access the volume.

We create the resources using `kubectl apply -f small-volume-example.yaml`, and thus create the pod and the PVC. This way, the a Longhorn Volume of size 10MiB gets dynamically provisioned. We can see that the block size in the filesystem we use is 4KiB.

```
root@testpod:/# df -h
Filesystem                Size  Used Avail Use% Mounted on
...
/dev/longhorn/<pvc-uuid>  5.4M   24K  5.2M   1% /mnt/longhorn
...
root@testpod:/# dumpe2fs /dev/longhorn/pvc-<pvc-uuid>
...
Block count:              2560
Block size:               4096
```

So, this Longhorn Volume consists of 2560 blocks, each of 4096 bytes (4KB). In Figure 3.2, we can see a depiction of this volume. Since the volume is new and no blocks have been written by the user, they appear empty.

As mentioned above, each replica consists of a chain of snapshots, the last of which is the volume's live state. In our case we have just created the volume and don't have any snapshots, so the live version is the only thing we can see inside the replica's directory. We can find this directory at the `/var/lib/longhorn/replicas` directory of the node.

```
user@k-master:/var/lib/longhorn/replicas$ sudo ls -la pvc-<pvc-uuid>
total 4868
drwx------ 2 root root     4096 Sep 24 10:41 .
drwxr-xr-x 7 root root     4096 Sep 24 10:41 ..
-rw------- 1 root root     4096 Sep 24 10:42 revision.counter
-rw-r--r-- 1 root root 10485760 Sep 24 10:42 volume-head-000.img
-rw-r--r-- 1 root root      126 Sep 24 10:41 volume-head-000.img.meta
-rw-r--r-- 1 root root      140 Sep 24 10:41 volume.meta
```

Notice that the file `volume-head-000.img` is the snapshot that holds the live state of the volume and its logical size is 10MB. However, if we try to find the actual size of the snapshot, we can see that it differs significantly.

```
user@k-master:/var/lib/longhorn/replicas$ sudo du --block-size 1 \
> pvc-<pvc-uuid>/volume-head-000.img
4964352 pvc-<pvc-uuid>/volume-head-000.img
```

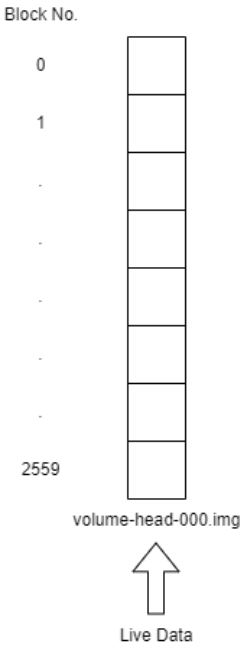


Figure 3.2: *A depiction of the new Longhorn Volume.*

According to the above, this snapshot is only 4964352 bytes or 4.73 MB. This happens because Longhorn replicas (and consequently snapshots) are built using Linux sparse files, which support thin provisioning.

If we try to write anything inside this volume, we will notice the actual size of the live version increase in size.

```
root@testpod:/mnt/longhorn# echo ”Hello, World!” > testfile
root@testpod:/mnt/longhorn# du -s -B 1 testfile
4096    testfile
...
user@k-master:/var/lib/longhorn/replicas$ sudo du --block-size 1 \
> pvc-<pvc-uuid>/volume-head-000.img
4968448 pvc-<pvc-uuid>/volume-head-000.img
```

Since we have now written to the volume, some blocks have been altered in Figure 3.3. These blocks are colored in the figure.

Now, if we take a snapshot of this volume using Longhorn UI, we will notice that a new snapshot has been created in `/var/lib/longhorn/replicas/pvc-<pvc-uuid>`. This snapshot has the same logical size as the previous live version, but now the live version has been reset and has zero size. Note also, that the live version now has a new name (`volume-head-001.img`).

```
user@k-master:/var/lib/longhorn/replicas$ sudo ls -la pvc-<pvc-uuid>/
total 4876
drwx------ 2 root root     4096 Sep 24 11:32 .
drwxr-xr-x 7 root root     4096 Sep 24 10:41 ..
```

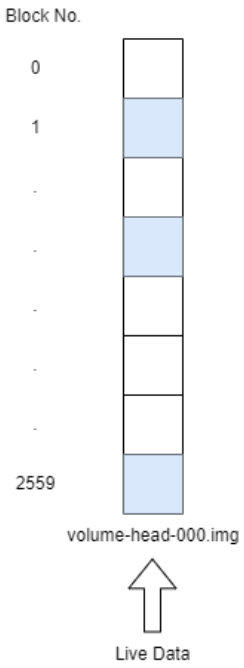


Figure 3.3: *The Longhorn Volume after writing data to it.*

```
-rw------- 1 root root     4096 Sep 24 11:26 revision.counter
-rw-r--r-- 1 root root 10485760 Sep 24 11:32 volume-head-001.img
-rw-r--r-- 1 root root      178 Sep 24 11:32 volume-head-001.img.meta
-rw-r--r-- 1 root root      192 Sep 24 11:32 volume.meta
-rw-r--r-- 1 root root 10485760 Sep 24 11:26 volume-snap-<snap-uuid>.img
-rw-r--r-- 1 root root      125 Sep 24 11:32 volume-snap-<snap-uuid>.img.meta
user@k-master:/var/lib/longhorn/replicas$ sudo du -s --block-size 1 \
pvc-<pvc-uuid>/volume-head-001.img
0       pvc-<pvc-uuid>/volume-head-001.img
user@k-master:/var/lib/longhorn/replicas$ sudo du -s --block-size 1 \
pvc-<pvc-uuid>/volume-snap-<snap-uuid>.img
4968448 pvc-<pvc-uuid>/volume-snap-<snap-uuid>.img
```

What has essentially happened, is that we kept the running live version as a snapshot and moved a pointer that points to the live data, to a new empty snapshot and held the previous live version as the snapshot. This process can be seen in Figure 3.4.

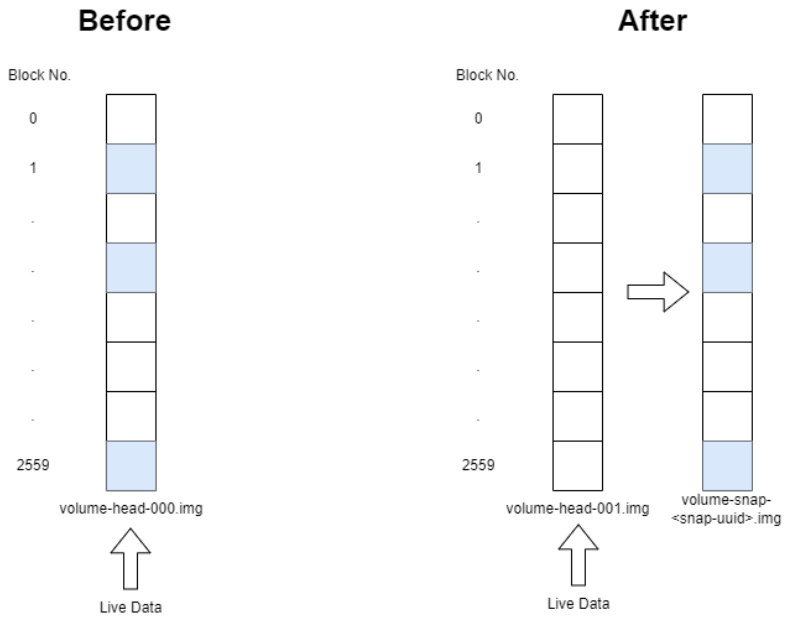


Figure 3.4: *The process of creating a new Longhorn Snapshot*

After creating some snapshots, the information that the volume contains is now shared among various snapshots. In case we want to read from the volume, we need to be able to find where the information we are looking for lies, without needing to start examining each version of the chain. This is why we maintain a ”Read Index” array which we fill lazily each time there is a ”Read” operation. The index resets for each block every time there is a ”Write” operation on that block, after which it points to the live data. Note that, we only maintain information on the source of the latest information of each block and not the history of the index (i.e. which version the index previously pointed to). This process is depicted in Figure 3.5.

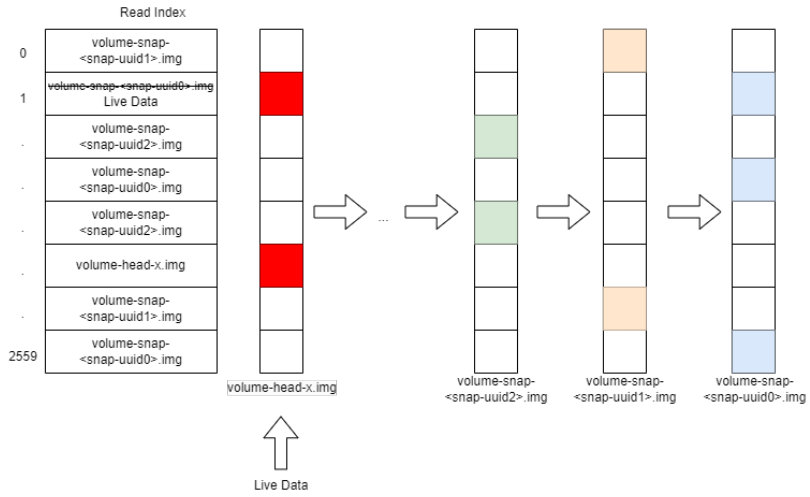


Figure 3.5: *The function of the read index.*

The read index is kept in memory and consumes one byte for each 4K block. The fact that the read index is byte-sized means that we can take a maximum of 254 snapshots for each volume. The read index consumes a certain amount of in-memory data structure for each replica. A 1 TB volume, for example, consumes 256 MB of in-memory read index. [53]

As mentioned above, Longhorn Snapshots are immutable, so they the user can't modify them. However, if the user deletes a snapshot that also has a subsequent one Longhorn will modify the next snapshot to merge the changes. For example, suppose that we have the following snapshots.

```
user@k-master:/var/lib/longhorn/replicas$ sudo ls -la pvc-<pvc-uuid>
total 5008
drwx------ 2 root root     4096 Sep 24 12:46 .
drwxr-xr-x 7 root root     4096 Sep 24 10:41 ..
-rw------- 1 root root     4096 Sep 24 12:46 revision.counter
-rw-r--r-- 1 root root 10485760 Sep 24 12:46 volume-head-003.img
-rw-r--r-- 1 root root      178 Sep 24 12:46 volume-head-003.img.meta
-rw-r--r-- 1 root root      192 Sep 24 12:46 volume.meta
-rw-r--r-- 1 root root 10485760 Sep 24 12:45 volume-snap-<snap-uuid1>.img
-rw-r--r-- 1 root root      177 Sep 24 12:45 volume-snap-<snap-uuid1>.img.meta
-rw-r--r-- 1 root root 10485760 Sep 24 12:46 volume-snap-<snap-uuid2>.img
-rw-r--r-- 1 root root      177 Sep 24 12:46 volume-snap-<snap-uuid2>.img.meta
-rw-r--r-- 1 root root 10485760 Sep 24 11:26 volume-snap-<snap-uuid0>.img
-rw-r--r-- 1 root root      125 Sep 24 11:32 volume-snap-<snap-uuid0>.img.meta
user@k-master:/var/lib/longhorn/replicas$ sudo du -sh \
> pvc-<pvc-uuid>/volume-snap-<snap-uuid1>.img
68K     pvc-<pvc-uuid>/volume-snap-<snap-uuid1>.img
user@k-master:/var/lib/longhorn/replicas$ sudo du -sh \
> pvc-<pvc-uuid>/volume-snap-<snap-uuid2>.img
56K     pvc-<pvc-uuid>/volume-snap-<snap-uuid2>.img
```

If we delete the snapshot called <snap-uuid1> through LonghornUI, we can notice that snapshot <snap-uuid2> has changed size.

```
user@k-master:/var/lib/longhorn/replicas$ sudo du -sh \
> pvc-<pvc-uuid>/volume-snap-<snap-uuid2>.img
100K    pvc-<pvc-uuid>/volume-snap-<snap-uuid2>.img
```

This process can also be seen in Figure 3.6.

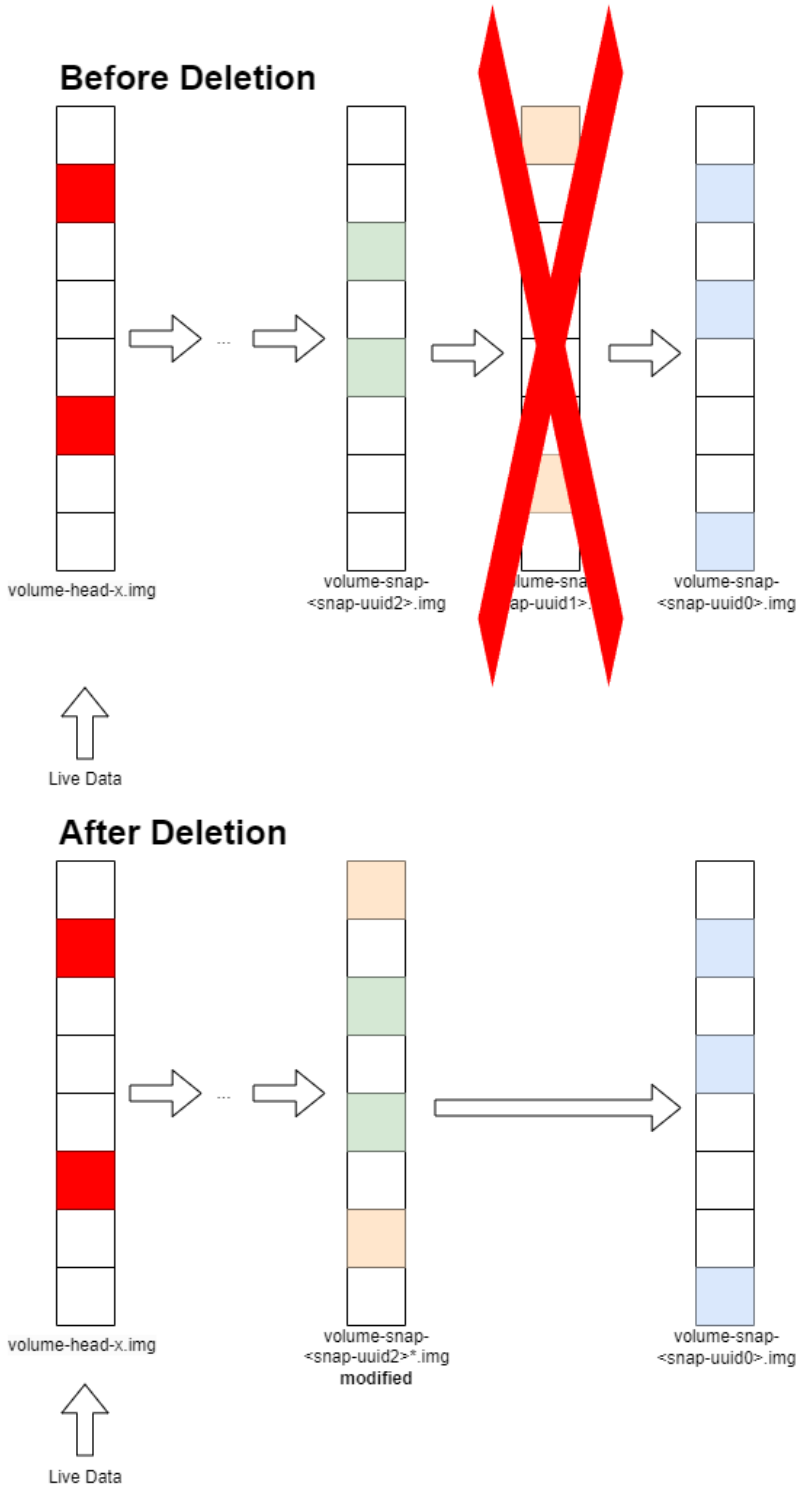


Figure 3.6: *The process of deleting an existing Longhorn Snapshot*

How Longhorn Backups work

A backup is an alternative way that Longhorn uses to hold the state of a volume. The basic difference of Longhorn Backup concept compared to Longhorn Snapshot is that the Backup is stored remotely outside of the cluster. [54] This is very useful, as they provide a form of secondary storage so that even if the cluster becomes unavailable, our data can still be retrieved. In Longhorn terms, the place where the backup is stored is called the Backupstore. This can be an NFS or S3 compatible object store external to the Kubernetes cluster. A Backup Target is the endpoint used to access a backupstore in Longhorn. [55] The user can create a Longhorn Backup through Longhorn UI.

When the user first creates a Longhorn Backup for a Longhorn Volume, Longhorn creates a Backup Volume. This is the backup that maps to one original volume, and it is located in the backupstore. The Backup Volume will contain multiple backups for the same volume.

A Longhorn Backup is created using one snapshot as a source, so that it reflects the state of the volume's data at the time that the snapshot was created. It can be thought of as a flattened version of a chain of snapshots. This means that when we take a backup of a volume, then only the latest version of each block it has will be kept, regardless of in which snapshot it can be found. Any block in the chain of snapshots that has been overwritten by a later snapshot will be lost for this specific backup.

An example of this process can be seen in Figure 3.7. Here we can see a chain of snapshots on the right side of the figure, and the Backupstore on the left side of the page. Colored blocks denote changed blocks compared to the previous snapshot. In the Backupstore, we have two backups: `backup-of-snap-2` and `backup-of-snap-3`. We can see that the blocks that we have backed up are present in the Backupstore and each backup maintains an array, where offsets are matched to blocks.

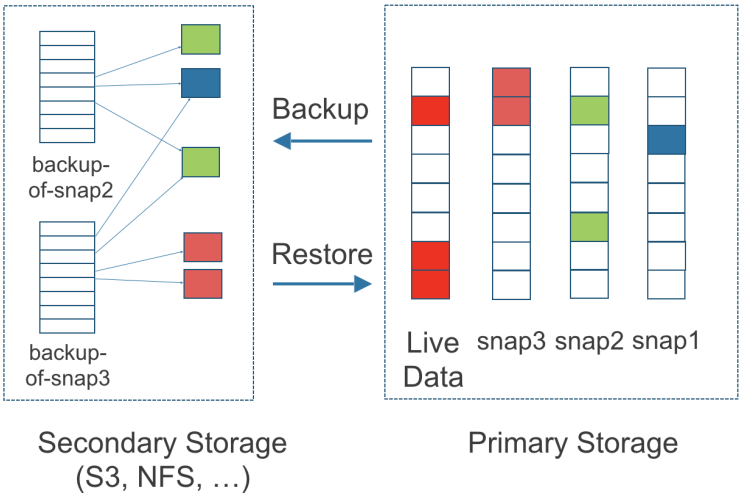


Figure 3.7: *How Longhorn Backups maintain differences from the previous one. [56]*

A Backup in the Backupstore is represented in the following way:

```
.
└── backupstore
    └── volumes
        ├── 29
        │   └── de
        │       └── pvc-<pvc-uuid>
        │           ├── backups
        │           │   └── backup_backup-<backup-id>.cfg
        │           ├── blocks
        │           │   ├── 34
        │           │   │   └── d6
        │           │   │       └── <block-checksum0>.blk
        │           │   ├── 73
        │           │   │   └── 18
        │           │   │       └── <block-checksum1>.blk
        │           │   ├── 94
        │           │   │   └── 43
        │           │   │       └── <block-checksum2>.blk
        │           │   └── d7
        .           │       └── 27
```

```
.               |               └── <block-checksum3>.blk
.               └── volume.cfg
```

Note that in the directory tree, there is a directory for each Backup Volume (in this case, `29/de/ pvc-<pvc-uuid>`). This directory contains the following items:

- A directory called `backups`, which contains the metadata files for all the Backups of the Backup Volume.
- A directory called `blocks`, which contains all the blocks that are maintained in this Backup.
- A file called `volume.cfg`, which contains the metadata for the Backup Volume.

An example of the `backup_backup-<backup-id>.cfg` file would be the following:

```
{
    "Name": "backup-<backup-id>",
    "VolumeName": "pvc-<pvc-uuid>",
    "SnapshotName": "<snapshot-uuid>",
    "SnapshotCreatedAt": "<datetime_snapshot_created>",
    "CreatedTime": "<datetime_backup_created>",
    "Size": "<size_in_bytes>",
    "Labels": {
        "KubernetesStatus": "{...}",
        "VolumeRecurringJobInfo": "{}",
        "longhorn.io/volume-access-mode": ""
    },
    "IsIncremental": "<true-or-false>",
    "CompressionMethod": "lz4",
    "ProcessingBlocks": {

    },
    "Blocks": [
        {
        "Offset": 0,
        "BlockChecksum": "<block-checksum0>"
        },
        {
        "Offset": 2097152,
        "BlockChecksum": "<block-checksum1>"
        },
        {
        "Offset": 4194304,
        "BlockChecksum": "<block-checksum2>"
        },
        {
        "Offset": 8388608,
        "BlockChecksum": "<block-checksum3>"
        }
    ],
    "SingleFile": {
```

```
        "FilePath": ""
    }
}
```

As we can see, the Backup metadata file contains information on names, datetimes, information on whether the backup is incremental or not (false for initial backups, true for all subsequent) and compression method. It also contains a map on which blocks are present in this volume and at which offset. These blocks can be found inside the `blocks` directory.

The `volume.cfg` file looks like the following:

```
{
    "Name": "pvc-<pvc-uuid>",
    "Size": "<size-in-bytes>",
    "Labels": {
        "KubernetesStatus": "{}",
        "VolumeRecurringJobInfo": "{}",
        "longhorn.io/volume-access-mode": ""
    },
    "CreatedTime": "<volume-created-datetime>",
    "LastBackupName": "backup-<backup-id>",
    "LastBackupAt": "<last-backup-datetime>",
    "BlockCount": "<number-of-blocks>",
    "BackingImageName": "",
    "BackingImageChecksum": "",
    "CompressionMethod": "lz4",
    "StorageClassName": "",
    "DataEngine": "v1"
}
```

Among the other information of the Backup Volume, in this file we can see the number of blocks that are present. This number of blocks can be shared across various backups, providing deduplication.

A Longhorn backup is essentially a configuration file that maps specific blocks to their offsets. These blocks are stored in the Backupstore for the volume, and with each new backup, any new blocks are added to the store.

Backups take advantage of the fact that each snapshot is a differencing file and only stores the changes in blocks from the last snapshot. This allows each new backup to also function incrementally, by only adding the changed blocks in the backupstore. For example, if there has been no change since the last backup, the new backup will take 0 bytes of space, although it essentially can be used to restore to the full data of the volume.

While snapshots can be hundreds of gigabytes -depending on the number of updated

data they contain compared to the previous snapshot-, each Longhorn Backup is made of files that are 2MB in size. To maintain the optimum balance between the number of files stored for each backup and the size of each file, Longhorn performs its operations using 2MB blocks. This way, if any of the 4K blocks in a 2MB boundary is changed, Longhorn will overwrite the entire 2MB block. This process is depicted in Figure 3.8.

In this Figure, we can see on the left side the initial backup of a volume. Each square is a 4kB block and a column of 4kB blocks (which share the same color), will be stored after the backup in the Backupstore as 2MB blocks. When a block changes (for example, the `Blk1` of the `A` -or white- row), in a subsequent backup Longhorn will backup the whole "column" of blocks, resulting in a new version (`A1`) of 2MB.

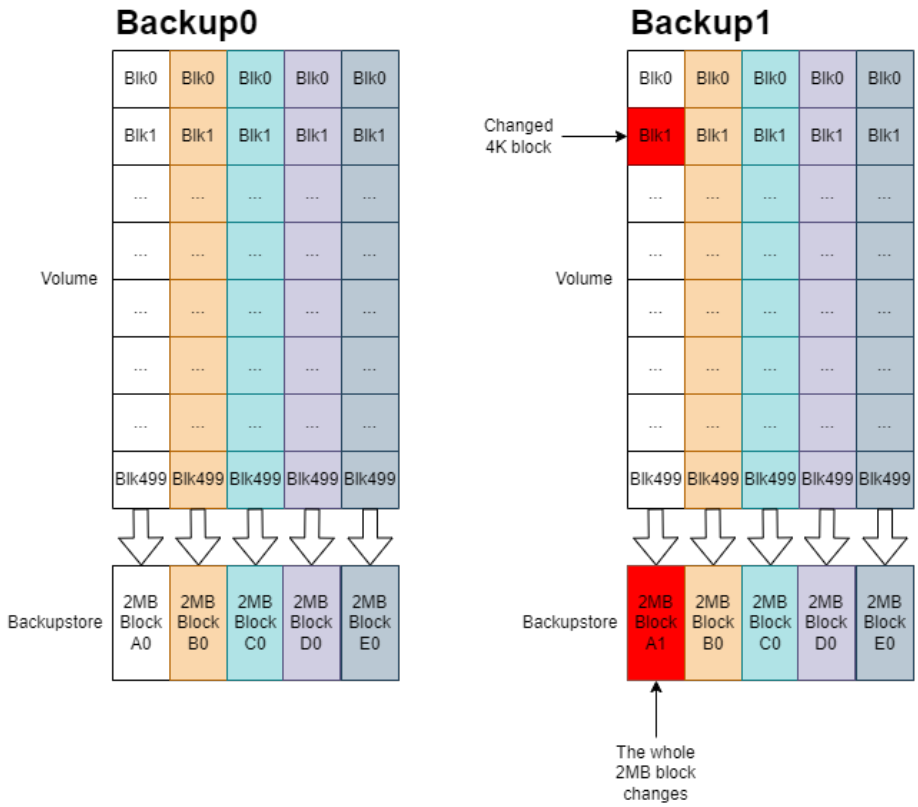


Figure 3.8: *How Longhorn Backups maintain differences from the previous one.*

Note that when a backup is deleted from the secondary storage, Longhorn does not delete all the blocks that it uses. Instead, it performs a garbage collection periodically to clean up unused blocks from secondary storage.

What the Longhorn Backing Image is

A QCOW2 or RAW image can be set as the backing/base image of a Longhorn volume, which allows Longhorn to be integrated with a VM. [57] We can get a backing image using the following sources:

- Download a backing image file (using a URL).
- Upload a file from our local machine. This option is available to Longhorn UI users.
- Export an existing in-cluster volume as a backing image.
- Restore a backing image from the backupstore.

When we try to back up the backing image of an existing Longhorn Volume, we need to have the Backup Target set, as the backup will live in the backupstore.

The use of checksums in Backing Images

Longhorn uses the checksum during restore, to validate the restored file. [58] The checksum of a backing image is the SHA512 checksum of the entire backing image file, not just its actual content. This means that when Longhorn computes the checksum of a qcow2 file, it treats the file as a raw file instead of using the qcow library to read the accurate content. In other words, users will always obtain the correct checksum by running shasum -a 512 <the file path>, regardless of the file format. The user should provide the expected SHA-512 checksum when creating a backing image based on the above. If there are issues during the preparation of the initial file, resulting in an incorrect checksum, Longhorn will treat the checksum of the produced file as the valid one. Consequently, this could lead to the backing image being unavailable. [59]

CSI Snapshot Detailed Sequence

The implementation of Velero CSI Support enables Velero to back up and restore CSI-backed volumes using the Kubernetes CSI Snapshot APIs, which is a Kubernetes API that provides the ability to take a snapshot of a persistent volume to protect against data

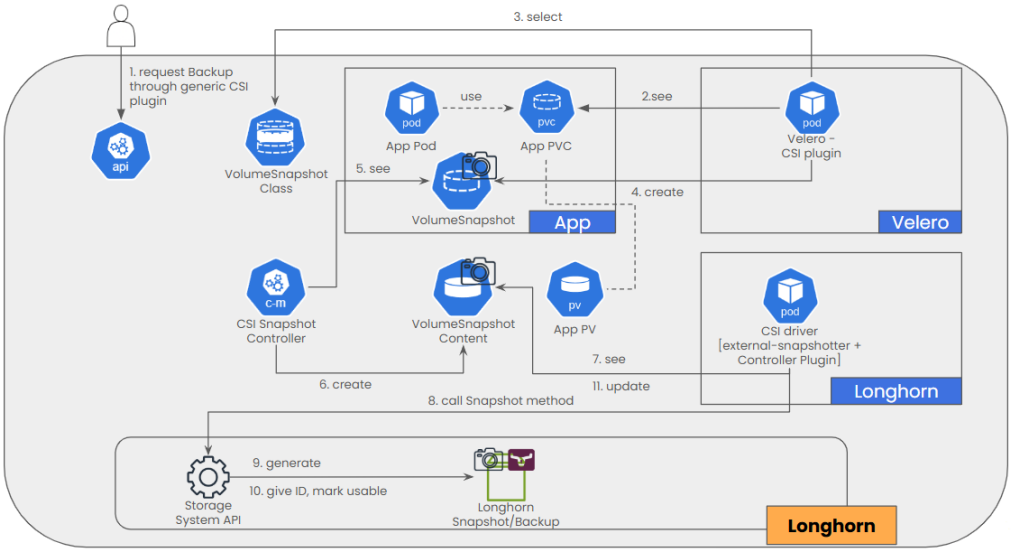


Figure 3.9: *The sequence of a Velero CSI snapshot.*

loss or data corruption. The procedure involves a collection of *BackupItemAction* plugins instead of relying on the Velero *VolumeSnapshotter* plugin interface. The detailed procedure is described in Figure 3.9.

1. [**User**] Request for a Backup.
   Using Velero cli: `velero backup create test-backup`
2. [**PVCBackupAction plugin**] Notice a PersistentVolumeClaim pointing to a PersistentVolume backed by a CSI Driver (Driver Name: ABC).
3. [**PVCBackupAction plugin**] Select the VolumeSnapshotClass with the following characteristics:
   - Has the same driver 'ABC' name as the driver name of the PV.
   - Has the label *velero.io/csi-volumesnapshot-class* set to true.
4. [**PVCBackupAction plugin**] Create a CSI VolumeSnapshot object with the PersistentVolumeClaim as a source.
5. [**CSI external-snapshotter controller**] See the VolumeSnapshot.
6. [**CSI external-snapshotter controller**] Create a VolumeSnapshotContent object, a cluster scoped resource that will point to the actual, disk-based snapshot in the storage system.
7. [**CSI driver**] See the newly created VolumeSnapshotContent resource.
8. [**CSI driver**] Call the storage system's APIs to generate the snapshot.

9. [**Storage system's API**] Generate the snapshot.
10. [**Storage system's API**] When the snapshot is completed, generate an ID and marks the snapshot as usable for restore.
11. [**CSI external-snapshotter controller**] Update the VolumeSnapshotContent object with a
`status.snapshotHandle` and sets the `status.readyToUse` field.

Using a VolumeSnapshotClass to create Longhorn CSI Snapshots

As shown above, the defined *VolumeSnapshotClass* is a major part of the CSI Snapshot procedure through Velero. Except for the fields described in section 2.2.4, the Longhorn specific parameters in the VolumeSnapshotClass are *type* and *export-type*. These parameters are necessary to create Longhorn CSI Snapshots depending on the use case. The *type* parameter can have 3 values, depending on our selection: *snap*, *bak* or *bi*. These parameters correspond to *Longhorn Snapshot*, *Longhorn Backup* and *Longhorn Backing Image*, depending on the type of the backup technique we want to use. The *export-type* parameter is only used if we have selected *bi* as our type. It can be either *raw* (default value) or *qcow2*, depending on the type of the backing/base image of the Longhorn volume we want to create.

VolumeSnapshotClass for CSI VolumeSnapshot Associated With Longhorn Snapshot

```
kind: VolumeSnapshotClass
apiVersion: snapshot.storage.k8s.io/v1
metadata:
name: longhorn-snapshot-vsc
driver: driver.longhorn.io
deletionPolicy: Delete
parameters:
type: snap
```

VolumeSnapshotClass for CSI VolumeSnapshot Associated With Longhorn Backup

```
kind: VolumeSnapshotClass
apiVersion: snapshot.storage.k8s.io/v1
metadata:
name: longhorn-backup-vsc
driver: driver.longhorn.io
deletionPolicy: Delete
parameters:
type: bak
```

VolumeSnapshotClass for CSI VolumeSnapshot Associated With Longhorn BackingImage

```
kind: VolumeSnapshotClass ssss
apiVersion: snapshot.storage.k8s.io/v1
metadata:
name: longhorn-snapshot-vsc
driver: driver.longhorn.io
deletionPolicy: Delete
parameters:
type: bi   # export-type default to raw if it is not given
export-type: qcow2
```

Since in our architecture we are using Longhorn as the default cloud-native distributed block storage and Longhorn is managed in Kubernetes via a CSI Plugin [60], we can use Velero to manage all Longhorn Volumes and create Snapshots / Backups for them. This means that the CSI Plugin allows us to create, restore and delete backups programmatically. [61]

Longhorn provides us with the ability to select between the three types of backup solutions described above: Longhorn Snapshot, Longhorn Backup or Longhorn Backing Image. All of these are represented by Custom Resource Definitions in Longhorn.

3.1.2 File System Backup

Velero has a mechanism implemented for backing up and restoring Kubernetes volumes attached to pods from the file system of the volumes. This is called `File System Backup (FSB)` or `PodVolumeBackup`. The use case of this is to be able to back up volumes that do not have a built-in backing up mechanism. This capability of Velero is provided by the integration of two open-source backup tools: `restic` and `kopia`.

File System Backup comes as an additional solution to the existing method of CSI Snapshots, which make use of each storage platform's CSI driver. Compared to the CSI snapshot, FSB has the following pros and cons.

Pros:

- File System Backup can be used to back up and restore almost any type of K8S volume. So, if there is no plugin available for the storage solution we are using, or we have Kubernetes volumes of alternative types like EFS, AzureFile, NFS, emptyDir, local, or any other volume type that doesn't have a native snapshot concept, we can use the File System Backup.

- It is not constrained to only one storage platform, and thus can be used to store the backup data to an alternative storage platform (e.g. a durable storage solution).

Cons:

- The backup operation is performed on the live file system while it is active, so objects selected for the backup job can be backed up at different points in time and some of the data may diverge over the course of the backup. This means that it is less consistent than a snapshot approach.
- Velero mounts the live file system on a hostpath directory to perform the File System Backup, so Velero Node Agent pods (that host the necessary components for FSB) need to run as root user and even under privileged mode in some environments.

As mentioned above, Velero uses either restic or kopia to perform File System Backup.

Restic Support

Restic is a backup program that can backup up files in many operating systems (Linux, BSD, Mac, Windows) to many different storage types (including self-hosted and online services) easily, effectively, securely, verifiably and freely. [62] It was the first of the two tools that were integrated with Velero to support File System Backup. Velero has integrated the Restic binary directly, so the operations are done by calling directly Restic commands.

Restic's Velero integration means that it is used both as a data mover for file system level data and a backup repository. However, it is considered incomplete and inefficient for the following reasons:

- Given that some components of File System Backup are also meant to be used in Velero Data Movement (see below for more info), Velero intends to use repository capabilities with various data movers. Restic is an inseparable unit made up of a file system data mover and a repository, therefore the repository capabilities are only available for Restic file system backup.

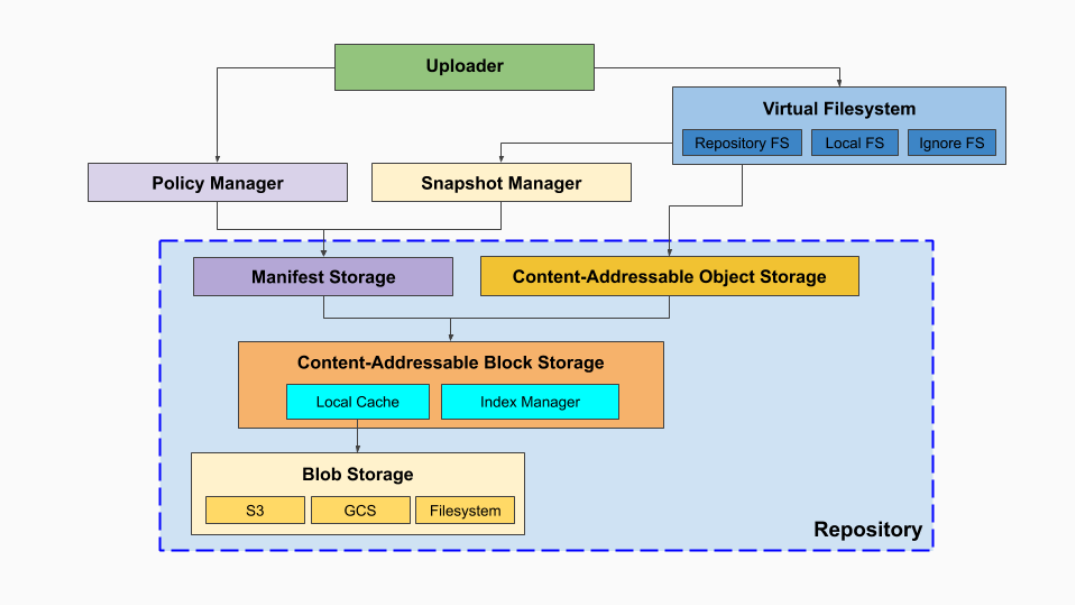


Figure 3.10: *The key components of Kopia.*

- The backup storage Velero supports through the Restic backup path depends on the storage Restic supports. As a result, if there is a requirement to introduce backup storage that Restic doesn't support, there is no way to make it.
- There is no way to enhance or extend the repository capabilities, because of the same reason – Restic is an inseparable unit, we cannot insert one or more customized layers to make the enhancements and extensions.

Moreover, Restic is reported by various users to have many performance issues on both the file system data mover side and the repository side. [63] The above reasons led to the need for the integration of a new backup tool for File System Backup, and that tool was selected to be Kopia.

Kopia Support

Kopia is an open-source backup/restore tool that allows us to create encrypted snapshots of our data and save the snapshots to remote or cloud storage of our choice, to network-attached storage or server, or locally on our machine. [64] Compared to Restic, Kopia's architecture divides modules more clearly according to their responsibilities and every module plays a complete role with clear interfaces. This makes it easier to take individual modules to Velero without losing critical functionalities.

Kopia offers the following features: [65]

- Backup files and directories using snapshots. Snapshots are maintained as a set of historical point-in-time records based on defined policies.
- Control what and how data is saved in snapshots using policies.
- Various options for the location where the snapshot data is saved. Snapshots can be saved to cloud, network or local storage. Many cloud platforms are supported, like Amazon S3 (or S3-compatible), Azure Blob Storage, Backblaze B2, Google Cloud Storage, any remote server or cloud storage that supports WebDAV, SFTP and some of the cloud storages supported by Rclone. The user can also self-host their own Kopia Repository Server.
- Restore snapshots using multiple methods. The user can mount the contents of a snapshot as a local disk , restore all data to any designated local or network location or selectively restore individual files.
- End-to-End 'zero knowledge' encryption. All data is encrypted before leaving the user's machine. Kopia also encyrpts both the content and names of backed up files/directories.
- Compression. Several compression methods are supported, including pgzip, s2 and zstd.
- Error correction using Reed-Solomon algorithm.
- Verifying backup validity and consistency. The user can request the verification of data consistency and validity inside the Kopia Repository in intervals defined by them.
- Recovering backed up data when there is data loss. Kopia has the ability in some cases to restore data in case of data loss in the Kopia Repository, unless there the actual backed up data file is corrupt.
- Regular automatic maintenance of repositories. Kopia runs automatic maintenance that ensures optimal performance and space usage.
- Caching. Kopia maintains a local cache of recently accessed objects making it possible to quickly browse repository contents without having to download from the storage location (regardless of whether the storage is cloud, network, or local).
- Both Command Line and graphical user interfaces.
- Optional server mode with API support to centrally manage backups of multiple machines.

- Speed. Based on benchmarks kopia is faster than restic. [66]

Velero integrates Kopia modules into Velero's code, mainly the two following modules:

- Kopia Uploader: Velero makes some wrap and isolation around it to create a generic file system uploader, which is used to backup pod volume data.
- Kopia Repository: Velero integrates it with Velero's Unified Repository Interface. It is used to preserve the backup data and manage the backup storage.

The Kopia Uploader is the entity responsible for moving the data to the Kopia Repository.

The Kopia Repository is the data structure where Kopia stores its data. [67] It holds all the tools that are necessary for the features that Kopia offers. The detailed architecture of the Kopia Repository is depicted in Figure 3.10.

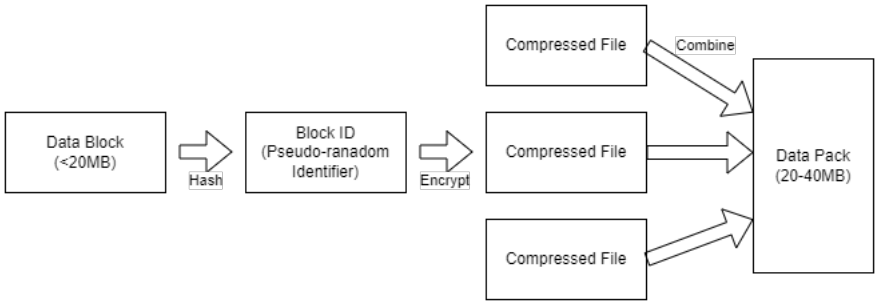


Figure 3.11: *The Content-Addressable Block Storage data packs.*

- Binary Large Object Storage (BLOB): BLOB storage is the place where the data is ultimately stored. [68] Kopia can use any storage solution that implements the Kopia Go storage interface. Many known cloud storage solutions are suitable as the Kopia Repository, as they provide high availability and data durability.
- Content-Addressable Block Storage (CABS): For Kopia to add some extra features that BLOB storage won't provide (for example encryption, deduplication), a new layer has been added. The CABS manages data blocks of small size (<20MB) and applies a cryptographic hash function (such as SHA2 or BLAKE2S) to them to produce a pseudo-random identifier (called the Block ID) which is used as the

name of the file. This serves as a deduplication technique, as data blocks with identical content will also have identical names. The data block is then encrypted (e.g. using AES256-GCM-HMAC-SHA256 or CHACHA20-POLY1305-HMAC-SHA256) an multiple encrypted data blocks are combined into larger data packs (20-40MB). Pack files have random names, concealing their contents and structure.

CABS also maintains an index mapping Block IDs to BLOB file names, offsets and lengths. In case this mapping is corrupted, a local index is also stored at the end of each data pack. CABS file names use prefixes to indicate their type:

  - `p` for data packs (e.g., pb4cf8ca179d71478fb8d4b00b79a9a72)
  - `q` for metadata packs (e.g., q7a9939814e8aba1fdda2d87965f324d3)
  - `x` for indices (e.g. xn_020db7984bd71c4042cea471a61fbcea1)

- Content-Addressable Object Storage (CAOS): While small objects can be stored directly as CABS blocks, larger objects first need to be split into smaller blocks. Similarly to the Block ID for CABS, the CAOS objects are assigned an Object ID, which is also content addressable. For smaller objects, these two are identical. Object IDs can have an optional single-letter prefix: [69]

  - `k` represents directory listing (e.g. kfed1b0498dc54d07cd69f272fb347ca3)
  - `m` represents manifest block (e.g. m0bf4da00801bd8c6ecfb66cffa67f32c)
  - `x` represents indirect JSON content (e.g. xac47f7ce280fdd81f04c670fec2353dc)

  When a CAOS object is a data block (e.g. doesn't have any of the prefixes above) , it is stored in a `p` data pack, while a CAOS object with prefixes is stored in a `q` metadata pack in the CABS layer.

  When dealing with blocks larger than the size of a single CABS block, Kopia combines multiple CABS blocks by using special indirect JSON content. The indirect content can be addressed by using the `I` prefix. So, the `x` prefix that represents indirect JSON content, combined with the `I` prefix give the actual content of a large file object.

- Label-Addressable Manifest Storage (LAMS): To make the accessing of storage objects easier for the user, Kopia uses Label-Addressable storage which is used to

persist small JSON objects called Manifests (describing snapshots, policies, etc). [70] These are identified by arbitrary key=value pairs called labels and they are internally stored as CABS blocks.

File System Backup Architecture in Velero

As mentioned earlier, Restic was the first tool to be utilized by Velero for File System Backup and Kopia came after it. This demanded a new and more effective design, since Velero integrates the Restic binary directly, but integrates the Kopia modules in the core code. To maintain backwards compatibility, there are now two paths for File System Backup: the (legacy) restic path and the kopia path. [4] The selection of the path used differs for the backup and restore process.

- For backup, the path used is specified at the installation time of Velero. The user needs to specify the `-uploader-type` flag, which can have two values: either `restic` or `kopia`. If the flag is not specified, the default value is `kopia`. The selected path cannot be changed after installation.
- For restore, the path is decided based on the path used to back up the volume. The PodVolumeBackup CR (described below) contains an `uploaderType` value, which is specified at Backup creation time. This is what determines the path.

For the File System Backup capabilities, Velero has developed three Custom Resources (CRs) and their associated controllers.

- Backup Repository. This CR represents the Backup Repository which is backed by either restic or kopia. The Backup Repository Controller is responsible for interacting with restic or kopia internally. Velero creates one Backup Repository per namespace.
- PodVolumeBackup. This CR represents a File System Backup of a volume that is attached to a pod. One instance of the PodVolumeBackup controller runs in each node of the cluster and that controller is responsible for all the pods in that node.
- PodVolumeRestore. This CR represents a restore from a File System Backup of a volume that is attached to a pod. Like the PodVolumeBackup controller, each

node in the cluster runs a controller for this resource (PodVolumeRestore controller).

FSB Detailed Sequence

In Figure 3.12, we can see the sequence followed when creating a new Backup or Restore. The detailed steps are the following.

1. [**User**] Request for a Backup/Restore.
   Using Velero cli: velero backup create –default-volumes-to-fs-backup test-bckp
2. [**Velero pod - Backup/Restore Controller**] Notice the new Backup/Restore.
3. [**Velero pod - PodVolume BR Manager**] Ensure a backup repository exists for the pod's namespace. If not, create it.
4. [**Velero pod - PodVolume BR Manager**] Create a PodVolumeBackup CR per volume.
5. [**Velero pod - PodVolume BR Manager**] Wait for the PodVolumeBackup resource to complete or fail.
6. [**Node Agent pod - PodVolume BR Controller**] Notice the new PodVolume CR and determine the path to be used.
   (a) For the Restic path, the Restic Uploader Provider invokes the existing Restic CLIs.
   (b) For the Kopia path, the Kopia Uploader Provider opens the unified repository for read/write, calls the Kopia Uploader to execute the backup, uses the Kopia shim as a Kopia repository interface implementation and finally reads/writes to/from the Kopia Repository.

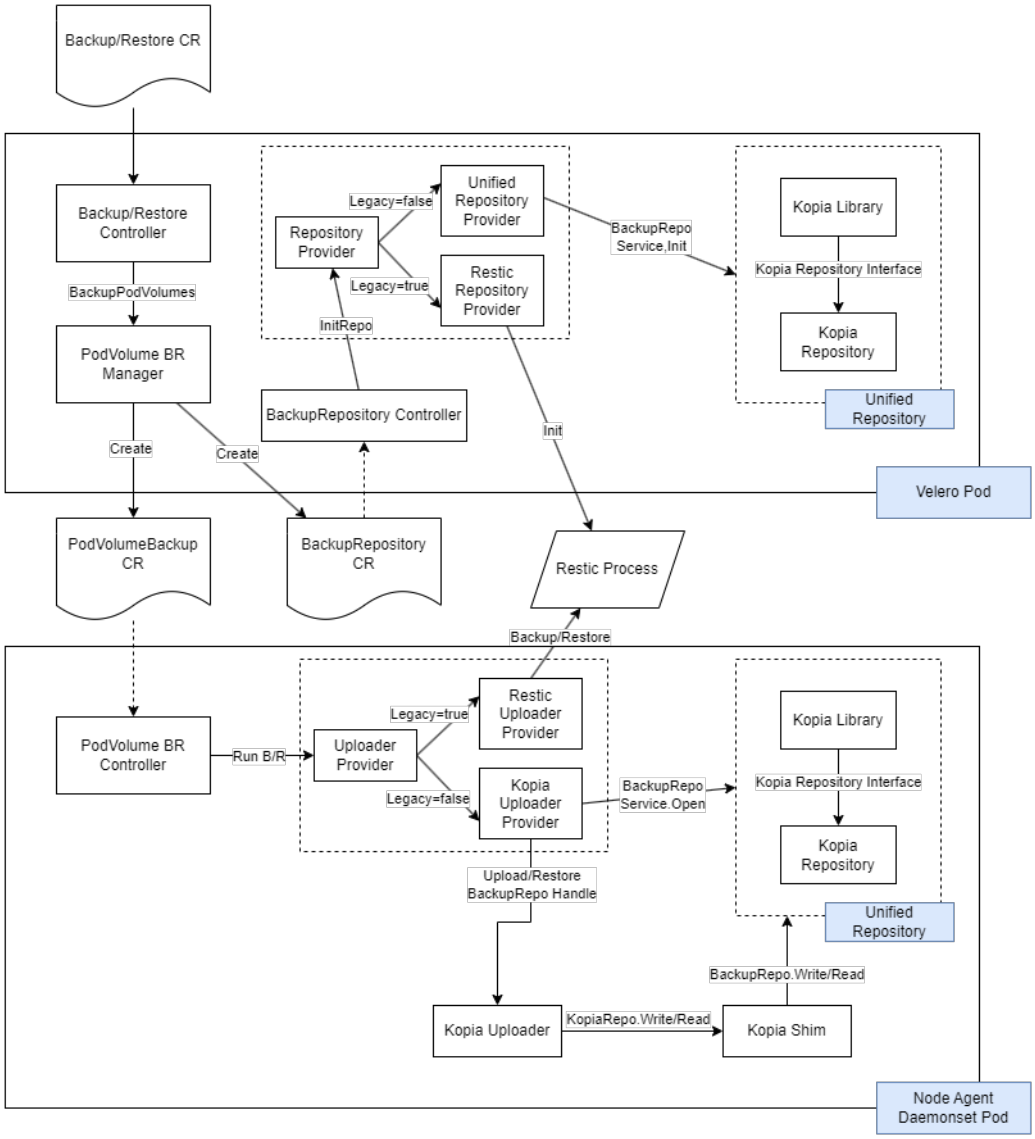


Figure 3.12: *The File System Backup process sequence. [63]*

For the Node Agent pod and the Kopia Uploader to be able to access the volume data, it has a `hostPath` volume mount of the `/var/lib/kubelet/pods` path of the host. We can make sure it exists.

```
user@k-master:/var/lib$ kubectl get pod  node-agent-6fkpm -n velero -o json |  jq -r '↩
     .spec.volumes[] | select(.hostPath)'
{"hostPath": {
    "path": "/var/lib/kubelet/pods",
    "type": ""},
 "name": "host-pods"}

{"hostPath": {
    "path": "/var/lib/kubelet/plugins",
    "type": ""},
  "name": "host-plugins"}
```

3.1.3 CSI Snapshot Data Movement

CSI Snapshot Data Movement is an alternative way to take backups. Velero has developed a new design, so that it takes first a CSI Snapshot and then it can access the snapshot data through a selection of a Data Movers and then back up the data to a backup storage connected to the Data Movers. [17] CSI Snapshot Data Movement can be useful in the following cases:

- For on-prem users: Many on-prem storage solutions don't offer durable snapshots. For example, Longhorn Snapshots data resides on the cluster. Using the CSI Snapshot Data Movement capability, the user can first take a CSI Snapshot and then move the data to a different/cheaper/more durable storage.

- For public cloud users: The Data Movement solution can be utilized to execute a multi cloud strategy. The user can take a snapshot using the snapshotting mechanisms of one provider, and then use a Data Mover to move this data to a different one. This allows Velero to function as the main tool for backup and restore even in a multi cloud setup.

As mentioned earlier, the File System Backup is also a solution that can be used to backup up the volume data to a pre-defined backup storage. However, using the CSI Snapshot Data Mover should always be preferred, as FSB reads data from the live PV. This makes the snapshot data less consistent.

The Velero CSI Snapshot Data Movement design supports both a built-in data mover and custom data movers. The Velero built-in data mover (VBDM) uses tools already integrated from the File System Backup design: the Kopia Uploader and the Kopia Repository. So, the VBDM is used to take a snapshot from a `PersistentVolume` and then send the snapshot files through the Kopia Uploader to the Kopia Repository. This is essentially done also with using File System Backup (with either Kopia or Restic as the provider), but using the Data Mover design:

1. Velero creates a CSI Snapshot of the PersistentVolume using a CSI driver.
2. Velero Built-in Data Mover creates a volume from the CSI snapshot.
3. VBDM transfers the data to the BackupStorageLocation using Kopia.
4. When the data transfer completes or any error happens, VBDM sets the DataUpload CR (a newly introduced CR) to the terminal state, either `Completed` or `Failed`.

Volume Snapshot Data Movement Design

For Velero to implement the new Velero Volume Snapshot Data Movement design, new CRs and modules needed to be introduced. The two new CRs are called `DataUpload` and `DataDownload` and they act as the protocol between data mover plugins and data movers.

Regarding the new modules, the `Data Mover Plugin (DMP)` is a new one, that is responsible for creating the DataUpload/DataDownload CRs and take the snapshot of the source volume. This also means that the DMP fully implements the Velero Backup Item Action/Restore Item Action (BIA/RIA) plugin interfaces, which is the way that the Velero pod communicates with the plugin. In the context of CSI Snapshot Data Movement, the CSI Plugin incorporates the DMP.

The other module that needed to be added, is the Data Mover itself. The Data Mover can either be the VBDM or a custom one. The VBDM consists of the following components:

- DataUpload/DataDownload (DU/DD) Controller: it watches for DU/DD CRs and when needed, it reconciles them and the data path to transfer data.

- Exposer: it exposes the snapshot/target volume as an endpoint recognizable by Velero generic data path.
- Velero Generic Data Path (VGDP): the VBDM uses the existing data movement path from the File System Backup design to write to / read from the Unified Repository (in this case, the Kopia Repository).
- Uploader: it is the module that reads data from the source and writes to backup repository for backup; while reads data from backup repository and writes to the restore target for restore. The VBDM uses the existing Kopia Uploader.

Velero uses the node agent daemonset to host these components.

VBDM Backup Detailed Sequence

In Figure 3.13, the Velero built-in Data Mover sequence is described. The detailed steps are the following.

1. [**Velero pod**] Velero delivers the PVC/PV to the corresponding DataMoverPlugin (here: the `CSI Plugin`) so that it takes the related actions to back it up.
   (a) [DataMover/CSI Plugin] Create a `VolumeSnapshot` CR in `SourceUser` namespace.
   (b) [DataMover/CSI Plugin] Create a `DataUpload` CR.
2. [**VBDM - DataUploadController**] Acquire Object Lock (make sure only one instance of DataUpload Controllers handle the CR).
3. [**VBDM - DataUploadController**] Request the exposer to expose the snapshot to be locally accessed and then wait until it is exposed.
4. [**VBDM - DataUploadController**] Release Object Lock.
5. [**VBDM - Exposer**] Expose the snapshot to the Velero Generic Data Path. For this, we need to have the following objects in the Velero namespace (visualized in Figure 3.14):
   (a) backupVSC: This is the Volume Snapshot Content object represents the CSI snapshot
   (b) backupVS: This the Volume Snapshot object for BackupVSC in Velero namespace

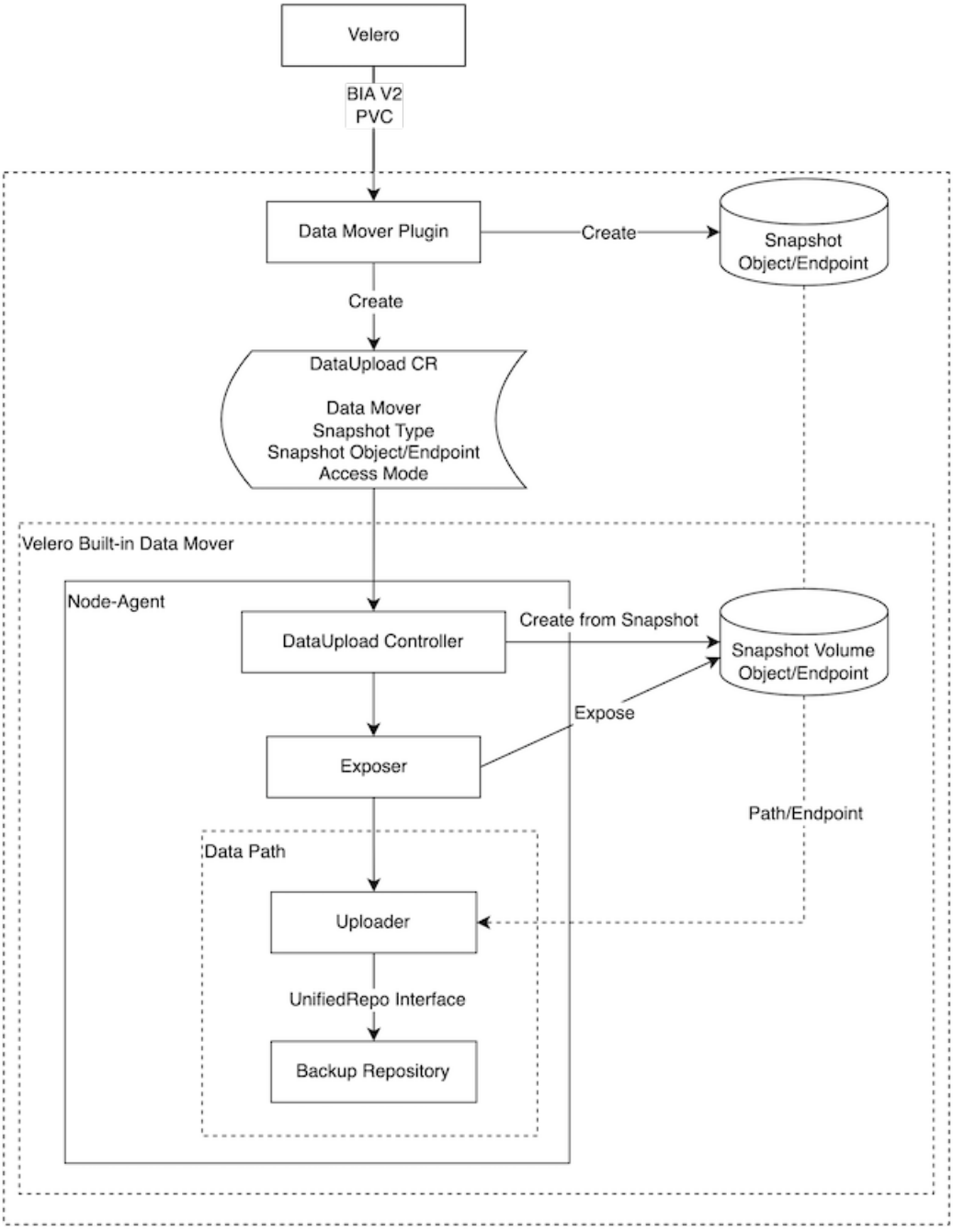


Figure 3.13: *The backup workflow during VBDM. [71]*

(c) backupPVC: This is the PVC created from the backupVS in Velero namespace. Specifically, backupPVC's data source points to backupVS

(d) backupPod: This is a pod attaching backupPVC in Velero namespace. As Kubernetes restriction, the PV is not provisioned until the PVC is attached to a pod and the pod is scheduled to a node. Therefore, after the backupPod is running, the backupPV which represents the data of the snapshot will be

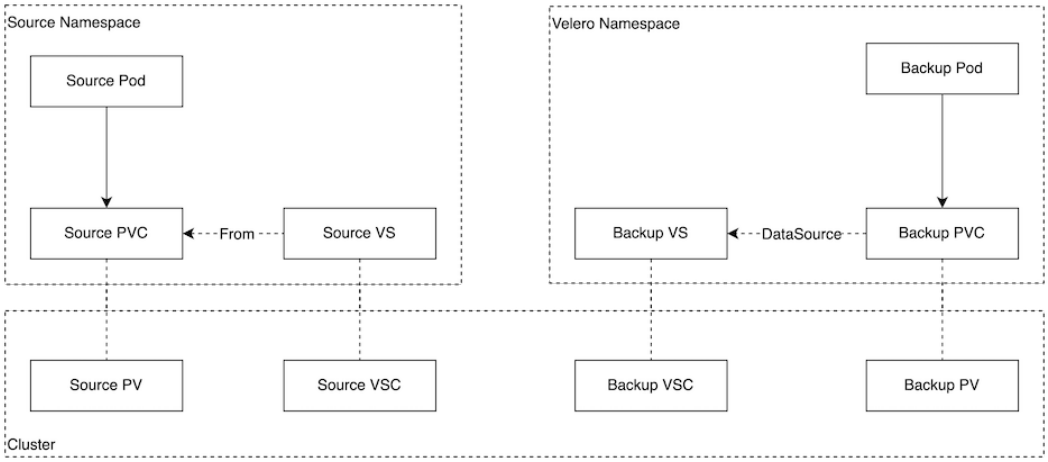


Figure 3.14: *The Velero built-in Data Mover resources created. [71]*

provisioned

(e) backupPV: This is the PV provisioned as a result of backupPod schedule, it has the same data of the snapshot

6. [**VBDM - DataUploadController**] VGDP can now access the snapshot data, so the controller calls the uploader to start the data backup to the Backup Repository.
7. [**VBDM - DataUploadController**] After the Backup is complete, the DataUpload Controller receives the Backup and updates the Repo Snapshot ID on the CR Status.

3.2 Restore Process with Velero

Using the Restore feature, the user can use Velero to restore all of the Kubernetes resources and volumes that were included in a previously created backup. Alternatively, Velero provides the user with the ability to filter the resources that will be restored based on various parameters, like namespace, resource type and selector labels. There is also the option to remap the namespace at which the resources belonging to a namespace belong, restoring them to a new namespace.

Velero will generally not delete any existing resources on the cluster, but the user can change this behavior using the `-existing-resource-policy`. The available values are `none`, which is the default value and `update`. Using `none`, Velero will skip the restoration of this resource, while using `update`, Velero will restore the version from the backup and use this instead of the existing version.

The default order in which Velero will restore the resources is the following: [72]

- Custom Resource Definitions
- Namespaces
- StorageClasses
- VolumeSnapshotClass
- VolumeSnapshotContents
- VolumeSnapshots
- PersistentVolumes
- PersistentVolumeClaims
- Secrets
- ConfigMaps
- ServiceAccounts
- LimitRanges
- Pods
- ReplicaSets
- Clusters
- ClusterResourceSets

If the user wants, they can edit this order using the `-restore-resource-priorities` flag.

3.2.1 Detailed Restore Process

The detailed steps Velero follows during a restore process are the following:

1. [**Velero Client**] Make a call to the Kubernetes API server to create a Restore object.
2. [**RestoreController**] Notice the new Restore object and perform validation.
3. [**RestoreController**] Fetch the backup information from the object storage service.
4. [**RestoreController**] Extract the tarball of backup cluster resources to the `/tmp` folder and performs some pre-processing on the resources. The preprocessing includes sorting the resources to the used restore order, discover the resources by their Kubernetes Group Version Resource (GVR), apply any configured resource filters and verify the target namespace (if the user has configured the `–namespace-mappings` name option).
5. [**RestoreController**] Begin restoring the eligible resources one at a time. Depending on the resource being restored, the RestoreController will need to apply modifications.
    - If the target namespace does not exist, it will create it.
    - If the resource is a PV, it will rename the PV and remap its namespace -if needed.
    - If the resource is a PVC, it will modify the PVC metadata -if needed.
    - If a RestoreItemAction plugin has been configured, the RestoreController will execute it.
    - If the user has enabled namespace remapping, the RestoreController will update the resource object's namespace.
    - The RestoreController adds a `velero.io/backup-name` label with the backup name and a `velero.io/restore-name` with the restore name to the resource. This can help the user easily identify restored resources and which backup they were restored from.
6. [**RestoreController**] Create the restore object on the target cluster. This is the point where it will restore the PV data with a different process depending on if it comes from a durable snapshot, File System Backup, or CSI snapshot.

7. [**RestoreController**] After the resource creation, the RestoreController might need to perform some post-processing.
    - If the resource is a Pod, the RestoreController will execute any Restore Hooks and wait for the hook to finish.
    - If the resource is a PV restored by File System Backup, the RestoreController waits for File System Backup's restore to complete.
    - If the resource is a Custom Resource Definition, the RestoreController waits for its availability in the cluster.

If any errors occur after completing these steps, the `RestoreController` will record the error in the restore result and proceed with the restoration process.

3.2.2 Persistent Volume Restoration

Depending on the origin of the PersistentVolume, the `RestoreController` will trigger a different process at Step 6.

PV backed up by Durable Snapshot

If the snapshot comes from a PV backed from a VolumeSnapshot plugin, the restoration process also uses the VolumeSnapshot plugin to create the new volume. Velero calls the plugins' interface to create a volume from a snapshot. The plugin returns the volume's volumeID. This ID is created by storage vendors and will be updated in the PV object created by Velero, so that the PV object is connected to the volume restored from a snapshot. [73]

PV backed up by File System Backup

When Velero finds a PersistentVolume that has been backed up by FSB, the following restore-specific actions take place. [74]

1. [**Velero Pod - Restore Controller**] Check each existing PodVolumeBackup custom resource in the cluster to backup from.

2. [**Velero Pod - PodVolume BR Manager**] For each PodVolumeBackup found, Velero ensures a backup repository exists for the pod's namespace.
3. [**Velero Pod - PodVolume BR Manager**] Add an init container to the pod, whose job is to wait for all FSB restores for the pod to complete.
4. [**Velero Pod - PodVolume BR Manager**] Create the pod, with the added init container, by submitting it to the Kubernetes API. Then, the Kubernetes scheduler schedules this pod to a worker node. If the pod fails to be scheduled for some reason (i.e. lack of cluster resources), the FSB restore will not be done.
5. [**Velero Pod - PodVolume BR Manager**] Create a PodVolumeRestore custom resource for each volume to be restored in the pod.
6. [**Velero Pod - Restore Controller**] The main Velero process now waits for each PodVolumeRestore resource to complete or fail.
7. [**Node Agent pod - PodVolume BR Controller**] Each PodVolumeRestore is handled by the controller on the appropriate node, which:
    - has a hostPath volume mount of /var/lib/kubelet/pods to access the pod volume data
    - waits for the pod to be running the init container
    - finds the pod volume's subdirectory within the above volume
    - based on the path selection, Velero invokes restic or kopia for restore
    - on success, writes a file into the pod volume, in a .velero subdirectory, whose name is the UID of the Velero restore that this pod volume restore is for
    - updates the status of the custom resource to Completed or Failed
8. [**Restored Pod - Init Container**] The init container that was added to the pod is running a process that waits until it finds a file within each restored volume, under `.velero` whose name is the UID of the Velero restore being run.
9. [**Restored Pod**] Once all such files are found, the init container's process terminates successfully and the pod moves on to running other init containers/the main containers.

PV backed up by CSI

When a PersistentVolume has been backed up using the CSI plugin, the restore will also be handled by the CSI plugin. At PVC restore time, the PVC resource created is

labeled with the annotation `velero.io/volume-snapshot-name` that has the name of the VolumeSnapshot object used for that restore. Velero also adds a `DataSource` field to the PVC that points to the VolumeSnapshot used.

CSI Snapshot Data Movement

When finding PVCs that are backed up using CSI Snapshot Data Movement, Velero follows the above sequence described in Subsection 3.2.1. Additionally, it performs the following.

1. [**Velero Pod - Restore Controller**] When it finds a PVC object, call CSI plugin through a Restore Item Action.
2. [**CSI plugin**] Check the backup information, if data movement was involved, create a DataDownload CR and then return to Velero restore.
3. [**Velero Pod - Restore Controller**] Continue to restore other resources, including other PVC objects.
4. [**Velero Pod - Restore Controller**] Periodically query the data movement status from CSI plugin. On the call, CSI plugin turns to check the phase of the DataDownload CRs.
5. [**Velero Pod - Restore Controller**] When all DataDownload CRs come to a terminal state (i.e., Completed, Failed or Cancelled), Velero restore will finish.
6. [**CSI plugin**] Expect the same data mover for the backup to handle the DataDownload CR. If no data mover was configured for the backup, Velero built-in data mover will handle it. If the DataDownload CR does not reach to the terminal state with in the given time, the DataDownload CR will be cancelled.
7. [**VBDM**] Create a volume with the same specification of the source volume.
8. [**VBDM**] Wait for Kubernetes to provision the volume. This may take some time varying from storage providers, but if the provision cannot be finished in a given time Velero built-in data mover will cancel this DataDownload CR.
9. [**VBDM**] After the volume is provisioned, start a data mover pod to transfer the data from the backup storage according to the backup storage location defined by users.
10. [**VBDM**] When the data transfer completes or any error happens, set the DataDownload CR to the terminal state (either Completed or Failed).

11. [**VBDM**] Monitor the cancellation request to the DataDownload CR. Once that happens, it cancels its ongoing activities, cleans up the intermediate resources and set the DataDownload CR to Cancelled.
12. [**VBDM**] Throughout the data transfer, monitor the status of the data mover pod and deletes it after DataDownload CR is set to the terminal state.

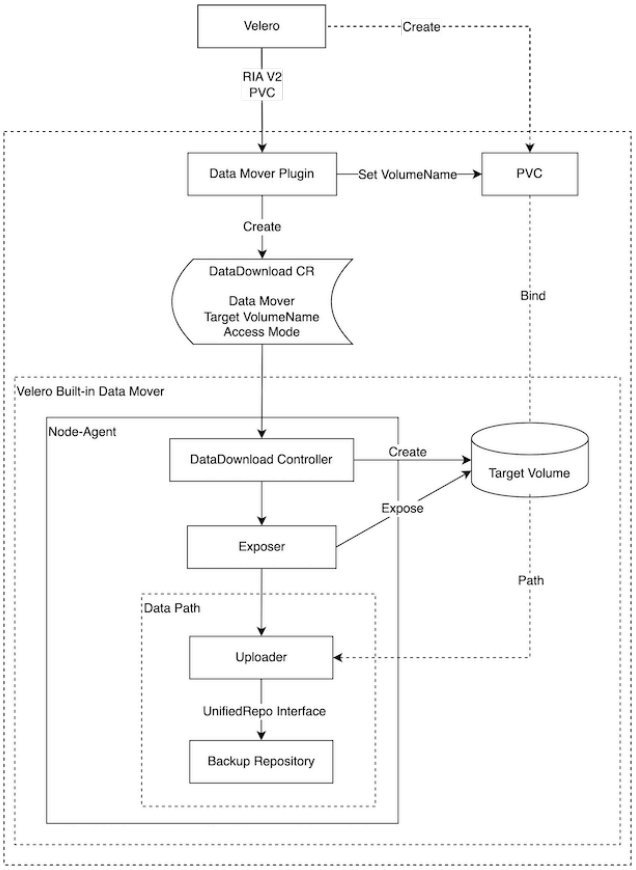


Figure 3.15: *The workflow of restore during VBDM.*

4 Implementation

In the previous chapters, we discussed the motivation for creating a Longhorn-native Velero plugin, we laid the foundations of knowledge for the development of this plugin and explained the inner mechanisms of all existing solutions when backing up a Longhorn volume using Velero. In this chapter, we will explain the implementation and deployment details of the plugin we created.

4.1 Overview

As mentioned in Chapter 2.6, Velero uses plugins to interact with different storage platforms. These plugins can be of different types, including Object Store, Volume Snapshotter, Backup Item Action, Restore Item Action and Delete Item Action. For the purpose of our diploma thesis we focused on the implementation of a Volume Snapshotter plugin.

4.2 Implementation

4.2.1 Velero Plugin architecture

A plugin for Velero typically follows the directory layout shown below.

```
.
├── CODE_OF_CONDUCT.md
├── CONTRIBUTING.md
├── Dockerfile
├── examples
│   └── with-pv.yaml
├── go.mod
├── go.sum
├── hack
│   └── build.sh
├── internal
│   └── plugin
│       ├── backupplugin.go
│       ├── backuppluginv2.go
│       ├── config.go
│       ├── deleteplugin.go
│       ├── helpers.go
│       ├── objectstoreplugin.go
│       ├── restoreplugin.go
│       ├── restorepluginv2.go
│       └── volumesnapshotterplugin.go
├── LICENSE
├── main.go
├── Makefile
├── _output
│   └── bin
│       └── linux
│           └── amd64
│               └── velero-plugin-longhorn
├── README.md
└── tilt-provider.json
```

Figure 4.1: *The directory layout of a Velero plugin.*

As we can see, there is a `main.go` file that registers the plugin and the actual implementation of the plugin. We can see the contents of such a file in Listing 4.1. In the main function of this file, we create a new `Server` resource, define the plugin server's command-line flags, register any plugin types we want to add to our plugin and finally run the plugin server. When we run the plugin server, we essentially create a `go-plugin`, which is Golang plugin system over RPC [75] created by Hashicorp. In particular for Velero plugins, it communicates over gRPC, and the HTTP2 protocol handles multiplexing. [76]

```go
package main

import (
    "github.com/sirupsen/logrus"
    "github.com/vmware-tanzu/velero-plugin-example/internal/plugin"
    "github.com/vmware-tanzu/velero/pkg/plugin/framework"
)

func main() {
    framework.NewServer().
        RegisterObjectStore("example.io/object-store-plugin", newObjectStorePlugin).
        RegisterVolumeSnapshotter("example.io/volume-snapshotter-plugin", ↩
            newNoOpVolumeSnapshotterPlugin).
        RegisterRestoreItemAction("example.io/restore-plugin", newRestorePlugin).
        RegisterRestoreItemActionV2("example.io/restore-pluginv2", newRestorePluginV2).
        RegisterBackupItemAction("example.io/backup-plugin", newBackupPlugin).
        RegisterBackupItemActionV2("example.io/backup-pluginv2", newBackupPluginV2).
        Serve()
}

func newBackupPlugin(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewBackupPlugin(logger), nil
}

func newBackupPluginV2(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewBackupPluginV2(logger), nil
}

func newObjectStorePlugin(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewFileObjectStore(logger), nil
}

func newRestorePlugin(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewRestorePlugin(logger), nil
}

func newRestorePluginV2(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewRestorePluginV2(logger), nil
}

func newNoOpVolumeSnapshotterPlugin(logger logrus.FieldLogger) (interface{}, error) {
    return plugin.NewNoOpVolumeSnapshotter(logger), nil
}
```

Listing 4.1: *The main file of a Velero plugin.*

4.2.2 Velero VolumeSnapshotter interface

In this diploma thesis, the aim was to develop a plugin that handles and adds extra steps to Volume Snapshots. Velero provides us with a VolumeSnapshotter interface for this, which our plugin needs to implement. This interface defines the operations needed by Velero to take snapshots of persistent volumes during backup, and to restore persistent volumes from snapshots during restore. [18]

```go
type VolumeSnapshotter interface {
    Init(config map[string]string) error

    CreateVolumeFromSnapshot(snapshotID, volumeType, volumeAZ string, iops *int64) (volumeID ↩
        string, err error)

    GetVolumeID(pv runtime.Unstructured) (string, error)

    SetVolumeID(pv runtime.Unstructured, volumeID string) (runtime.Unstructured, error)

    GetVolumeInfo(volumeID, volumeAZ string) (string, *int64, error)

    CreateSnapshot(volumeID, volumeAZ string, tags map[string]string) (snapshotID string, err ↩
        error)

    DeleteSnapshot(snapshotID string) error
}
```

Listing 4.2: *The VolumeSnapshotter interface.*

Each operation need to be implemented by a function and the job they perform is described below.

- `Init` prepares the VolumeSnapshotter for usage using the provided map of configuration key-value pairs. It returns an error if the VolumeSnapshotter cannot be initialized from the provided config.
- `CreateVolumeFromSnapshot` creates a new volume in the specified availability zone, initialized from the provided snapshot, and with the specified type and IOPS (if using provisioned IOPS).
- `GetVolumeID` returns the cloud provider specific identifier for the PersistentVolume.
- `SetVolumeID` sets the cloud provider specific identifier for the PersistentVolume.
- `GetVolumeInfo` returns the type and IOPS (if using provisioned IOPS) for the specified volume in the given availability zone.
- `CreateSnapshot` creates a snapshot of the specified volume, and applies the provided set of tags to the snapshot.
- `DeleteSnapshot` deletes the specified volume snapshot.

It is not necessary that all of these methods are implemented in the same way for each plugin, as different storage solutions have different features and each plugin could be configured to have a different result.

4.2.3 Velero Plugin for Longhorn implementation

In our implementation, we configured the `main.go` file to register a VolumeSnapshotter plugin, as seen in Listing 4.3.

```
1  package main
2
3  import (
4      "github.com/sirupsen/logrus"
5      "github.com/spf13/pflag"
6      "github.com/thomaspant/velero-plugin-longhorn/internal/plugin"
7      veleroplugin "github.com/vmware-tanzu/velero/pkg/plugin/framework"
8  )
9
10 func main() {
11     veleroplugin.NewServer().
12         BindFlags(pflag.CommandLine).
13         RegisterVolumeSnapshotter("thomaspant/velero-plugin-longhorn", ↩
               newLonghornVolumeSnapshotterPlugin).
14         Serve()
15
16 }
17
18 func newLonghornVolumeSnapshotterPlugin(logger logrus.FieldLogger) (interface{}, error) {
19     return plugin.NewLHVolumeSnapshotter(logger), nil
20 }
```

Listing 4.3: *The main.go file of velero-plugin-longhorn.*

The logic of our plugin is included in the `volumesnapshotterplugin.go` file. This files contains the implementations of all the functions needed for a VolumeSnapshotter plugin.

First of all, we define some constants that represent the names of the arguments that the user will need to provide at Velero installation time for the `VolumeSnapshotLocation`.

```
const (
    backuptargetKey     = "backup-target-url"
    backupsecretKey     = "backup-target-secret"
    lhendpointKey       = "longhorn-endpoint"
    dstsecretKey        = "dst-secret"
)
```

Listing 4.4: *The keys defined for VolumeSnapshotLocation.*

As we can see, the user needs to input values for "backup-target-url", "backup-target-secret", "longhorn-endpoint" and "dst-secret".

Then we create a new struct called `LHVolumeSnapshotter`. This struct contains the state for the snapshotter and is the pointer receiver of all the functions implemented in the plugin.

```
// LHVolumeSnapshotter is a plugin for containing state for the blockstore
type LHVolumeSnapshotter struct {
    log             logrus.FieldLogger
    rancherclient   *lhclient.RancherClient
    backuptarget    string
    backupsecret    string
    lhendpoint      string
    dstsecret       string
}
```

Listing 4.5: *The LHVolumeSnapshotter struct.*

We also need a function that returns a new LHVolumeSnapshotter.

```
// NewLHVolumeSnapshotter instantiates a LHVolumeSnapshotter.
func NewLHVolumeSnapshotter(logger logrus.FieldLogger) *LHVolumeSnapshotter {
    return &LHVolumeSnapshotter{log: logger}
}
```

Listing 4.6: *The NewLHVolumeSnapshotter function.*

We can now start implementing the methods needed for the VolumeSnapshotter interface. The first is the `Init` method.

```
func (p *LHVolumeSnapshotter) Init(config map[string]string) error {
    // Log that the Init method was called with the provided configuration.
    p.log.Infof("Init called with config %v", config)

    // Validate that the necessary configuration keys are present in the config map.
    if err := veleroplugin.ValidateVolumeSnapshotterConfigKeys(config, backuptargetKey, ↩
        backupsecretKey, lhendpointKey, dstsecretKey); err != nil {
        p.log.Infof("Error with keys!")
```

```
        return err
    }

    // Assign values from the config map to the struct's fields.
    p.backuptarget = config[backuptargetKey]
    p.backupsecret = config[backupsecretKey]
    p.lhendpoint = config[lhendpointKey]
    p.dstsecret = config[dstsecretKey]

    // Ensure that all required configuration values are not empty, returning an error if any are missing.
    if p.backuptarget == "" {
        return errors.Errorf("missing %s in longhorn configuration", p.backuptarget)
    }
    if p.backupsecret == "" {
        return errors.Errorf("missing %s in longhorn configuration", p.backupsecret)
    }
    if p.lhendpoint == "" {
        return errors.Errorf("missing %s in longhorn configuration", p.lhendpoint)
    }
    if p.dstsecret == "" {
        return errors.Errorf("missing %s in longhorn configuration", p.dstsecret)
    }

    // Create a new Rancher client using the Longhorn endpoint URL.
    rc, err := lhclient.NewRancherClient(&lhclient.ClientOpts{
        Url: p.lhendpoint,
    })

    // Log and return if there is an error creating the Rancher client.
    if err != nil {
        p.log.Infof("Error creating Rancher Client")
        return err
    }

    // Store the Rancher client in the struct.
    p.rancherclient = rc

    // Create a new setting for the backup target.
    newurl := &lhclient.Setting{
        Name:  "backup-target",
        Value: p.backuptarget,
    }

    // Retrieve the current backup target setting.
    bupurl, err := p.rancherclient.Setting.ById("backup-target")

    // Log and return if there is an error getting the backup target setting.
    if err != nil {
        p.log.Infof("Error getting setting backup-target", err)
        return err
    }

    // Update the backup target setting with the new URL.
    updurl, err := p.rancherclient.Setting.Update(bupurl, newurl)

    // Log and return if there is an error updating the backup target setting.
    if err != nil {
        p.log.Infof("Error updating settings", err)
        return err
    }

    // Log the updated backup target URL.
    p.log.Infof("New backup target URL: ", updurl)

    // Create a new setting for the backup credential secret.
    newsecret := &lhclient.Setting{
        Name:  "backup-target-credential-secret",
        Value: p.backupsecret,
    }

    // Retrieve the current backup credential secret setting.
    bupsecret, err := p.rancherclient.Setting.ById("backup-target-credential-secret")

    // Log and return if there is an error getting the backup credential secret.
    if err != nil {
        p.log.Infof("Error getting backup target credential secret", err)
        return err
    }

    // Update the backup credential secret setting.
    updsecret, err := p.rancherclient.Setting.Update(bupsecret, newsecret)

    // Log and return if there is an error updating the credential secret.
    if err != nil {
        p.log.Infof("Error updating settings", err)
        return err
    }

    // Log the updated backup credential secret.
    p.log.Infof("New secret: ", updsecret)

    // Return nil to indicate the initialization was successful.
    return nil
```

```
}
```

Listing 4.7: *The Init method.*

In this method we execute the following actions.

- We use the `ValidateVolumeSnapshotterConfigKeys` function to ensure that the user has provided us with the arguments that are necessary for the VolumeSnapshotLocation config and make sure that their content is not empty.
- We create a client to communicate with Longhorn.
- Given that the user has provided the backup target URL and the backup target secret, during the Init function we set them in Longhorn. This way, the user doesn't have to use the Longhorn UI to set the backup target.

The next method we implement is `CreateVolumeFromSnapshot`. This method is used to create a new volume during restore.

```
func (p *LHVolumeSnapshotter) CreateVolumeFromSnapshot(snapshotID, volumeType, volumeAZ string, iops *int64) (string, error) {
    // Log that the method was called with the provided snapshot ID.
    p.log.Infof("CreateVolumeFromSnapshot called", snapshotID)

    // Initialize variables for the volume ID, backup URL, volume name, and size.
    var volumeID string
    backupurl := ""
    backupvolname := ""
    backupsize := ""

    // List all backup volumes using the Rancher client.
    volumes, err := p.rancherclient.BackupVolume.List(nil)
    p.log.Infof("List of Backup Volumes: ", volumes)

    // Log if there is an error while listing backup volumes.
    if err != nil {
        p.log.Infof("Error listing volumes")
    }

    // Log the configured backup target.
    p.log.Infof("Backup Target: ", p.backuptarget)

    // Iterate over the backup volumes to find the specified snapshot.
    for volumeitem := range volumes.Data {
        // List all backups for the current volume.
        backupList, err := p.rancherclient.BackupVolume.ActionBackupList(&volumes.Data[volumeitem])

        // Log an error if backups for the volume cannot be listed.
        if err != nil {
            p.log.Infof("Error listing backups for Volume ", volumes.Data[volumeitem].Name)
        }

        // Search for the backup that matches the given snapshot ID.
        for backupitem := range backupList.Data {
            if backupList.Data[backupitem].SnapshotName == snapshotID {
                backupurl = backupList.Data[backupitem].Url
                backupvolname = backupList.Data[backupitem].VolumeName
                backupsize = backupList.Data[backupitem].Size
            }
        }
    }

    // If no backup URL was found, log that the snapshot was not found and return.
    if backupurl == "" {
        p.log.Infof("The specified snapshot was not found")
        return "", nil
    }

    // Generate a random identifier for the new volume's name.
    temp := uuid.New().String()[:6]
    p.log.Infof("Name of the new Volume: ", temp)
```

```
52
53      // Define the new volume's configuration, using the backup URL and size from the snapshot.
54      newvolumedesc := lhclient.Volume{
55          AccessMode: "rwo",
56          FromBackup: backupurl,
57          Name:       backupvolname,
58          Size:       backupsize,
59      }
60
61      // Create the new volume with the specified configuration.
62      newvolume, err := p.rancherclient.Volume.Create(&newvolumedesc)
63
64      // Log and return if there is an error creating the volume.
65      if err != nil {
66          p.log.Infof("Error creating Volume", err)
67      }
68
69      // Log details of the newly created volume.
70      p.log.Infof("New Volume: ", newvolume)
71
72      // Set the volume ID to the name of the newly created volume.
73      volumeID = newvolume.Name
74
75      // Return the new volume ID and nil error if successful.
76      return volumeID, nil
77  }
```

Listing 4.8: *The CreateVolumeFromSnapshot method.*

In this method we execute the following:

- First, we need to find the Longhorn Backup that we will use for the Volume restore. Since Longhorn doesn't provide us with a defined method to find the Longhorn Backup from the Longhorn Snapshot ID, we iterate over all backup volumes and backups to find the one that matches with the Snapshot ID provided.
- Now that we have the necessary configuration, we can create a new Volume that comes from the specified backup.
- Finally, we return the ID of the new volume.

Next, we have the `GetVolumeInfo` method.

```
1  func (p *LHVolumeSnapshotter) GetVolumeInfo(volumeID, volumeAZ string) (string, *int64, error)↩
       {
2      // Log that the GetVolumeInfo method was called with the given volume ID and AZ.
3      p.log.Infof("GetVolumeInfo called", volumeID, volumeAZ)
4
5      // Retrieve volume information by its ID using the Rancher client.
6      vol, err := p.rancherclient.Volume.ById(volumeID)
7
8      // If there is an error fetching the volume, return an error wrapped with stack trace.
9      if err != nil {
10         return "", nil, errors.WithStack(err)
11     }
12
13     // Log the frontend type of the volume, indicating how it is exposed (e.g., block device ↩
           or iSCSI).
14     p.log.Infof("Frontend of Volume: ", vol.Frontend)
15
16     // Return the frontend type of the volume, a nil value for iops (not needed here), and nil↩
           for error.
17     return vol.Frontend, nil, nil
18 }
```

Listing 4.9: *The GetVolumeInfo method.*

This method is generally used to provide the type and IOPS of the specified Volume, but since Longhorn doesn't support provisioned IOPS we return `nil` for this value. Re-

garding the type of the Volume, we return the frontend type of the Longhorn Volume -block device or iSCSI.

Then we proceed to the `IsVolumeReady` method.

```
func (p *LHVolumeSnapshotter) IsVolumeReady(volumeID, volumeAZ string) (ready bool, err error){
    // Log that the IsVolumeReady method was called with the given volume ID and AZ.
    p.log.Infof("IsVolumeReady called", volumeID, volumeAZ)

    // Retrieve volume details by its ID using the Rancher client.
    vol, err := p.rancherclient.Volume.ById(volumeID)

    // Log an error if the volume cannot be retrieved by ID.
    if err != nil {
        p.log.Infof("Error getting volume by ID")
    }

    // Log the readiness status of the volume, indicating if it is ready for use.
    p.log.Infof("Is Volume ready: ", vol.Ready)

    // Return the readiness status of the volume and nil error if successful.
    return vol.Ready, nil
}
```

Listing 4.10: *The IsVolumeReady method.*

This method returns whether the Volume is ready or not.

The next and possibly most important method is `CreateSnapshot`. This is the method where we create the new Snapshot and perform any additional logic.

```
func (p *LHVolumeSnapshotter) CreateSnapshot(volumeID, volumeAZ string, tags map[string]string) (string, error) {
    // Log the creation of a snapshot with specified volume ID, AZ, and tags.
    p.log.Infof("CreateSnapshot called", volumeID, volumeAZ, tags)

    // Retrieve the volume details using the Rancher client.
    myvolume, err := p.rancherclient.Volume.ById(volumeID)
    if err != nil {
        p.log.Infof("Error getting Volume %s", volumeID)
    }

    // Generate a unique ID for the snapshot.
    tempuuid := uuid.New().String()
    p.log.Infof("Name of the new Snapshot: ", tempuuid)

    // Create a snapshot of the volume and store snapshot details.
    snapshotinfo, err := p.rancherclient.Volume.ActionSnapshotCreate(myvolume, &lhclient.SnapshotInput{
        Name: "snapshot-" + tempuuid,
    })

    if err != nil {
        p.log.Infof("Error creating snapshot: %s", err)
    }

    p.log.Infof("Volume snapshot created", snapshotinfo)

    // Backup the snapshot.
    backupinfo, err := p.rancherclient.Volume.ActionSnapshotBackup(myvolume, &lhclient.SnapshotInput{
        Name: "snapshot-" + tempuuid,
    })

    p.log.Infof("Volume backup created", backupinfo)

    if err != nil {
        p.log.Infof("Error creating backup: %s", err)
    }

    isBackupReady := false
    snapshotname := "snapshot-" + tempuuid
    p.log.Infof("Snapshot Name: ", snapshotname)

    // Get backup information by ID and verify backup readiness.
    backupVolId := backupinfo.Name
    p.log.Infof("Backup Info: ", backupVolId)

    backupVol, err := p.rancherclient.BackupVolume.ById(backupVolId)

    if err != nil {
```

```
48          p.log.Infof("Error getting BackupVolume: %s", err)
49      }
50
51      p.log.Infof("Backup Volume: ", backupVol)
52
53      // List backup volumes and check for backup readiness.
54      volumeBackupList, err := p.rancherclient.BackupVolume.ActionBackupList(backupVol)
55
56      if err != nil {
57          p.log.Infof("Error getting BackupList: %s", err)
58      }
59
60      p.log.Infof("Backup Volume List: ", volumeBackupList)
61
62      var backupId string
63      var finalBackup *lhclient.Backup
64
65      // Identify the backup ID corresponding to the snapshot.
66      for backupitem1 := range volumeBackupList.Data {
67          p.log.Infof("Temp BackupID: ", volumeBackupList.Data[backupitem1].Id)
68          if volumeBackupList.Data[backupitem1].SnapshotName == snapshotname {
69              backupId = volumeBackupList.Data[backupitem1].Id
70              p.log.Infof("BackupID: ", backupId)
71          }
72      }
73
74      // Poll the backup state until it is marked "Completed."
75      for !isBackupReady {
76          finalBackup, err = p.rancherclient.BackupVolume.ActionBackupGet(backupVol, &lhclient.↩
                BackupInput{
77              Name: backupId,
78          })
79
80          if err != nil {
81              p.log.Infof("Error getting backup: %s", err)
82          }
83
84          if finalBackup.State == "Completed" {
85              p.log.Infof("Backup is ready, condition is true!")
86              isBackupReady = true
87          } else {
88              p.log.Infof("Backup for snapshot ", finalBackup.SnapshotName, "is not ready yet!")
89              p.log.Infof("The state of the backup is ", finalBackup.State)
90              time.Sleep(1 * time.Second)
91          }
92      }
93
94      // Create Kubernetes client configuration.
95      kubeconfig, err := rest.InClusterConfig()
96
97      if err != nil {
98          p.log.Fatalf("Error building in-cluster config: %s", err.Error())
99      }
100
101     p.log.Infof("Kubeconfig: ", kubeconfig)
102
103     // Initialize Kubernetes client.
104     clientset, err := kubernetes.NewForConfig(kubeconfig)
105
106     if err != nil {
107         p.log.Fatalf("Error creating clientset: %s", err.Error())
108     }
109
110     p.log.Infof("Clientset: ", clientset)
111
112     // Retrieve source and destination secrets from Kubernetes.
113     namespace := "longhorn-system"
114     srcsecret, err := clientset.CoreV1().Secrets(namespace).Get(context.TODO(), p.backupsecret↩
            , metav1.GetOptions{})
115
116     if err != nil {
117         p.log.Fatalf("Error retrieving secret: %s", err.Error())
118     }
119
120     p.log.Infof("Source Secret: ", srcsecret)
121
122     dstsecret, err := clientset.CoreV1().Secrets(namespace).Get(context.TODO(), p.dstsecret, ↩
            metav1.GetOptions{})
123
124     if err != nil {
125         p.log.Fatalf("Error retrieving secret: %s", err.Error())
126     }
127
128     p.log.Infof("Destination Secret: ", dstsecret)
129
130     // Retrieve and parse MinIO endpoint and access information from secrets.
131     srcminioendpoint := getHostPart(string(srcsecret.Data["AWS_ENDPOINTS"]))
132     srcaccessKeyID := string(srcsecret.Data["AWS_ACCESS_KEY_ID"])
133     srcsecretAccessKey := string(srcsecret.Data["AWS_SECRET_ACCESS_KEY"])
134     srcuseSSL := string(srcsecret.Data["USESSL"])
135     p.log.Infof("Endpoint, accessKey, AccessSecret, UseSSL:", srcminioendpoint, srcaccessKeyID↩
            , srcsecretAccessKey, srcuseSSL)
136
```

```
137     dstminioendpoint := getHostPart(string(dstsecret.Data["AWS_ENDPOINTS"]))
138     dstaccessKeyID := string(dstsecret.Data["AWS_ACCESS_KEY_ID"])
139     dstsecretAccessKey := string(dstsecret.Data["AWS_SECRET_ACCESS_KEY"])
140     dstuseSSL := string(dstsecret.Data["USESSL"])
141     p.log.Infof("Endpoint, accessKey, AccessSecret, UseSSL:", dstminioendpoint, dstaccessKeyID↩
            , dstsecretAccessKey, dstuseSSL)
142
143     // Retrieve the bucket name and location from the backup target URL.
144     bucketName, location := getBucketName(p.backuptarget)
145     p.log.Infof("BucketName: ", bucketName)
146     p.log.Infof("BucketLocation: ", location)
147
148     // Parse SSL settings for MinIO clients.
149     srcuseSSLfix, err := strconv.ParseBool(srcuseSSL)
150
151     if err != nil {
152         p.log.Infof("Error parsing string as bool: %s", err)
153     }
154
155     dstuseSSLfix, err := strconv.ParseBool(dstuseSSL)
156
157     if err != nil {
158         p.log.Infof("Error parsing string as bool: %s", err)
159     }
160
161     // Create MinIO clients for source and destination.
162     srcminioClient, err := minio.New(srcminioendpoint, &minio.Options{
163         Creds:  credentials.NewStaticV4(srcaccessKeyID, srcsecretAccessKey, ""),
164         Secure: srcuseSSLfix,
165     })
166
167     if err != nil {
168         p.log.Infof("Error creating MinIO client 1", err)
169     }
170
171     dstminioClient, err := minio.New(dstminioendpoint, &minio.Options{
172         Creds:  credentials.NewStaticV4(dstaccessKeyID, dstsecretAccessKey, ""),
173         Secure: dstuseSSLfix,
174     })
175
176     if err != nil {
177         p.log.Infof("Error creating MinIO client 2", err)
178     }
179
180     // Initialize context for bucket operations.
181     ctx := context.Background()
182     p.log.Infof("Context: ", ctx)
183
184     // Check if the source bucket exists and log if it doesn't.
185     exists1, err := srcminioClient.BucketExists(ctx, bucketName)
186
187     if err != nil {
188         p.log.Infof("Error getting BucketExists", err)
189     }
190
191     p.log.Infof("", exists1)
192
193     if !exists1 {
194         p.log.Infof("Error! The source bucket doesn't exist!")
195     }
196
197     // Create destination bucket if it doesn't exist.
198     exists2, err := dstminioClient.BucketExists(ctx, bucketName)
199
200     if err != nil {
201         p.log.Infof("Error getting BucketExists", err)
202     }
203
204     if !exists2 {
205         err = dstminioClient.MakeBucket(ctx, bucketName, minio.MakeBucketOptions{Region: ↩
                location})
206
207         if err != nil {
208             p.log.Infof("Failed to create destination bucket: %v", err)
209         }
210
211         p.log.Infof("Destination bucket '%s' created.\n", bucketName)
212     }
213
214     // List and sync objects from source to destination bucket.
215     objectCh := srcminioClient.ListObjects(ctx, bucketName, minio.ListObjectsOptions{
216         Recursive: true,
217     })
218
219     for object := range objectCh {
220         if object.Err != nil {
221             p.log.Infof("Error while listing objects: %v", object.Err)
222             continue
223         }
224
225         p.log.Infof("Syncing object: %s\n", object.Key)
226
227         // Download and upload each object.
```

```
228         srcObject, err := srcminioClient.GetObject(ctx, bucketName, object.Key, minio.↩
                GetObjectOptions{})
229
230         if err != nil {
231             p.log.Infof("Failed to download object '%s': %v", object.Key, err)
232             continue
233         }
234
235         uploadInfo, err := dstminioClient.PutObject(ctx, bucketName, object.Key, srcObject, ↩
                object.Size, minio.PutObjectOptions{
236             ContentType: object.ContentType,
237         })
238
239         if err != nil {
240             p.log.Infof("Failed to upload object '%s' to destination: %v", object.Key, err)
241             continue
242         }
243
244         p.log.Infof("Successfully synced: %s (size: %d)\n", uploadInfo.Key, uploadInfo.Size)
245     }
246
247     // Return the snapshot ID after creation.
248     p.log.Infof("CreateSnapshot returning with ", snapshotinfo.Id)
249     return snapshotinfo.Id, nil
250 }
```

Listing 4.11: *The CreateSnapshot method.*

This method executes the following.

- We generate a name for the new Snapshot using a UUID.
- We create a new Snapshot for the specified volume.
- We create a new Backup from the newly created Snapshot (since each Backup in Longhorn comes from a Snapshot).
- We wait until the new Backup comes in "Completed" state.
- We retrieve the necessary secrets for the Backup Target and the destination Bucket that exist in the "longhorn-system" namespace.
- We get the necessary values from these secrets (minioendpoint, accessKeyID, secretAccessKey, useSSL).
- We create MinIO clients for source and destination buckets (creating the destination bucket if it doesn't exist).
- We synchronize all data from the source bucket to the destination bucket (this data includes the latest backup data).
- We return the new Snapshot ID.

As we can see in the code of `CreateSnapshot` method, we use two helper functions: `getBucketName` and `getHostPart`. Let's explain what they do.

```
1 func getBucketName (s string) (string, string) {
2     // Split the string by "://"
3     parts := strings.Split(s, "://")
4     if len(parts) < 2 {
5         return "", "" // Invalid input
6     }
7
8     // Split the second part by "@"
9     bucketRegion := strings.Split(parts[1], "@")
```

```
    if len(bucketRegion) < 2 {
        return "", "" // Invalid input
    }

    // Extract bucket name and region
    bucketName := bucketRegion[0]    // First part is the bucket name
    region := strings.TrimSuffix(bucketRegion[1], "/")  // Remove the trailing slash from the region

    return bucketName, region
}
```

Listing 4.12: *The getBucketName function.*

As we can see in Listing 4.12, the getBucketName function gets as input a string of a Backup Target endpoint of the form `s3://bucket_name@location/` and gives the bucket name and the location in string form.

Then we have the `getHostPart` function.

```
func getHostPart (input string) string {
    // Check if the string starts with "http://" or "https://"
    if !strings.HasPrefix(input, "http://") && !strings.HasPrefix(input, "https://") {
        input = "http://" + input
    }

    // Parse the URL
    parsedURL, err := url.Parse(input)
    if err != nil {
        fmt.Println("Error parsing URL:", err)
        return ""
    }

    return parsedURL.Host
}
```

Listing 4.13: *The getHostPart function.*

This function gets an input string which contains a URL and removes the leading `http://` or `https://` if it exists. The purpose of this function is to be able to use the URL with the MinIO client, regardless of how the user has used it as input to the secret.

The next method is `DeleteSnapshot`.

```
func (p *LHVolumeSnapshotter) DeleteSnapshot(snapshotID string) error {
    // Log the initiation of snapshot deletion with snapshotID
    p.log.Infof("DeleteSnapshot called", snapshotID)

    // List all available volumes
    listvol, err := p.rancherclient.Volume.List(nil)
    if err != nil {
        p.log.Infof("Error listing volumes")
    }

    // Log the list of volumes retrieved
    listvoldata := listvol.Data
    p.log.Infof("Volume List Data: ", listvoldata)

    // Variable to store the index of the volume containing the snapshot
    var volumeindex int
    p.log.Infof("Length of Volume List Data: ", len(listvoldata))

    // Iterate through the volumes to find the specified snapshot
    for index := range listvoldata {
        // Attempt to get the snapshot with snapshotID for the current volume
        snapget, err := p.rancherclient.Volume.ActionSnapshotGet(&listvol.Data[index], &lhclient.SnapshotInput{
            Name: snapshotID,
        })

        if err != nil {
            p.log.Infof("Error getting snapshot")
        } else {
```

```
            // Snapshot found; store the volume index and break the loop
            volumeindex = index
            p.log.Infof("Volume Snapshot found: ", snapget)
            break
        }

        p.log.Infof("Snapshot not found")
    }

    // Log the index of the volume containing the snapshot
    p.log.Infof("Volume Index of the Snapshot: ", volumeindex)

    // Attempt to delete the snapshot from the volume at volumeindex
    snapdel, err := p.rancherclient.Volume.ActionSnapshotDelete(&listvol.Data[volumeindex], &lhclient.SnapshotInput{
        Name: snapshotID,
    })

    if err != nil {
        p.log.Infof("Error deleting snapshot")
    }

    p.log.Infof("Snapshot deleted: ", snapdel)

    // Retrieve the backup volume associated with the volume
    bupvol, err := p.rancherclient.BackupVolume.ById(listvol.Data[volumeindex].Name)
    if err != nil {
        p.log.Infof("Error getting BackupVolume: ", err)
    }

    // Attempt to delete the backup of the snapshot
    bupdel, err := p.rancherclient.BackupVolume.ActionBackupDelete(bupvol, &lhclient.BackupInput{
        Name: snapshotID,
    })

    if err != nil {
        p.log.Infof("Error deleting Backup: ", err)
    }

    p.log.Infof("Backup deleted: ", bupdel)

    return nil
}
```

Listing 4.14: *The DeleteSnapshot function.*

Given a snapshot ID, this method deletes the specified snapshot.

Then we have the `GetVolumeID` method.

```
func (p *LHVolumeSnapshotter) GetVolumeID(unstructuredPV runtime.Unstructured) (string, error) {
    // Log the initiation of the GetVolumeID function with the unstructured persistent volume
    p.log.Infof("GetVolumeID called", unstructuredPV)

    // Create a new PersistentVolume object to store the converted data
    pv := new(v1.PersistentVolume)

    // Convert the unstructured PV data into a typed PersistentVolume object
    if err := runtime.DefaultUnstructuredConverter.FromUnstructured(unstructuredPV.UnstructuredContent(), pv); err != nil {
        return "", errors.WithStack(err)
    }

    // Check if the CSI (Container Storage Interface) specification is present
    if pv.Spec.CSI == nil {
        return "", errors.New("unable to retrieve CSI Spec")
    }

    // Check if the VolumeHandle is set in the CSI specification
    if pv.Spec.CSI.VolumeHandle == "" {
        return "", errors.New("unable to retrieve Volume handle")
    }

    // Log the VolumeHandle that will be returned
    p.log.Infof("Returning: ", pv.Spec.CSI.VolumeHandle)

    return pv.Spec.CSI.VolumeHandle, nil
}
```

Listing 4.15: *The GetVolumeID function.*

In this method given an unstructured PersistentVolume object, we convert it into a typed PV and if it is a CSI PV, we return this Volume's handle.

The final method we have implemented is `SetVolumeID`.

```
func (p *LHVolumeSnapshotter) SetVolumeID(unstructuredPV runtime.Unstructured, volumeID string) (runtime.Unstructured, error) {
    // Log the invocation of SetVolumeID with the provided unstructuredPV and volumeID
    p.log.Infof("SetVolumeID called with unstructuredPV %v and volumeID", unstructuredPV, volumeID)

    // Create a new PersistentVolume instance
    pv := new(v1.PersistentVolume)

    // Convert the unstructuredPV to a typed PersistentVolume object
    if err := runtime.DefaultUnstructuredConverter.FromUnstructured(unstructuredPV.UnstructuredContent(), pv); err != nil {
        // Return an error if the conversion fails
        return nil, errors.WithStack(err)
    }

    // Check if the CSI spec is present in the PersistentVolume
    if pv.Spec.CSI == nil {
        // Return an error if the CSI spec is not found
        return nil, errors.New("Spec.CSI.VolumeHandle not found")
    }

    // Set the VolumeHandle in the CSI spec to the provided volumeID
    pv.Spec.CSI.VolumeHandle = volumeID

    // Convert the updated PersistentVolume back to an unstructured format
    res, err := runtime.DefaultUnstructuredConverter.ToUnstructured(pv)
    if err != nil {
        return nil, errors.WithStack(err)
    }

    return &unstructured.Unstructured{Object: res}, nil
}
```

Listing 4.16: *The SetVolumeID function.*

Similarly to the `GetVolumeID`, this method converts this input unstructured PersistentVolume object into a typed PV and then sets its ID to the given string.

4.3 Deployment

4.3.1 Deploying the Plugin

As mentioned above, Velero plugins implement the `go-plugin` system. In Velero context, they are containers that are part of the main Velero pod that exists in the Velero namespace. So, to use this custom plugin we need first to build the image and then deploy it. For this cause, we can use the Makefile provided by Velero.

```
PKG := github.com/thomaspant/velero-plugin-longhorn
BIN := velero-plugin-longhorn

REGISTRY ?= velero
IMAGE    ?= $(REGISTRY)/velero-plugin-longhorn
VERSION  ?= main

GOOS   ?= $(shell go env GOOS)
GOARCH ?= $(shell go env GOARCH)

# local builds the binary using 'go build' in the local environment.
.PHONY: local
```

```
local: build-dirs
    CGO_ENABLED=0 go build -v -o _output/bin/$(GOOS)/$(GOARCH) .

# test runs unit tests using 'go test' in the local environment.
.PHONY: test
test:
    CGO_ENABLED=0 go test -v -timeout 60s ./...

# ci is a convenience target for CI builds.
.PHONY: ci
ci: verify-modules local test

# container builds a Docker image containing the binary.
.PHONY: container
container:
    docker build -t $(IMAGE):$(VERSION) .

# push pushes the Docker image to its registry.
.PHONY: push
push:
    @docker push $(IMAGE):$(VERSION)
ifeq ($(TAG_LATEST), true)
    docker tag $(IMAGE):$(VERSION) $(IMAGE):latest
    docker push $(IMAGE):latest
endif

# modules updates Go module files
.PHONY: modules
modules:
    go mod tidy -compat=1.17

# verify-modules ensures Go module files are up to date
.PHONY: verify-modules
verify-modules: modules
    @if !(git diff --quiet HEAD -- go.sum go.mod); then \
        echo "go module files are out of date, please commit the changes to go.mod and go.sum"↩
            ; exit 1; \
    fi

# build-dirs creates the necessary directories for a build in the local environment.
.PHONY: build-dirs
build-dirs:
    @mkdir -p _output/bin/$(GOOS)/$(GOARCH)

# clean removes build artifacts from the local environment.
.PHONY: clean
clean:
    @echo "cleaning"
    rm -rf _output
```

Listing 4.17: *The Makefile of the plugin.*

Using `IMAGE=<repo>/<name> VERSION=<tag> make container`, we can build the image. This invokes the `docker build -t $(IMAGE):$(VERSION) .` command. Then we need to push it to a registry that is accessible by our cluster. For this cause, we used Dockerhub and pushed the images using the `docker push $(IMAGE):$(VERSION)` command.

Now that our image is built and available in an accessible registry, we need to add the plugin to our Velero deployment. We can do this during Velero installation time or when Velero is already deployed.

If we do it at installation time, we must include our plugin at the `--plugins` argument. Additionally, we must add the parameters necessary for the VolumeSnapshotLocation config. We can do this using the `--snapshot-location-config` command-line argument. An example of such an installation could be the following.

```
velero install --provider aws     \
       --plugins <object_store_plugin>,thomaspant/velero-plugin-longhorn:latest     \
```

```
        --bucket <bucket_name>      \
        --secret-file <credentials_file>     \
        --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=<s3_url> \
        --snapshot-location-config backup-target-url=$BACKUPTARGETURL,backup-target-secret=↩
            $BACKUPTARGETSECRET,longhorn-endpoint=$LHENDPOINT,dst-secret=$DSTSECRET
```

Listing 4.18: *The velero install command.*

After installing Velero, we need to change the `spec.provider` field in VolumeSnapshotLocation to the name of our plugin -in this case `thomaspant/velero-plugin-longhorn`.

```
apiVersion: velero.io/v1
kind: VolumeSnapshotLocation
metadata:
    creationTimestamp: <timestamp>
    generation: 1
    labels:
    component: velero
    name: default
    namespace: velero
    resourceVersion: <version>
    uid: <uid>
spec:
    config:
        backup-target-secret: <secret-name>
        backup-target-url: <url>
        dst-secret: <destination-secret-name>
        longhorn-endpoint: <endpoint>
    provider: thomaspant/velero-plugin-longhorn
```

Listing 4.19: *The VolumeSnapshotLocation needed for the plugin.*

Alternatively, if we want to do this after having installed Velero and without removing it, we can add the plugin on the fly and then edit the existing VolumeSnapshotLocation. Velero client provides us with the ability to do this, using the following command. After doing this, we also need to edit the VolumeSnapshotLocation to match Listing 4.19.

```
velero plugin add thomaspant/velero-plugin-longhorn:latest
kubectl edit volumesnapshotlocation -n velero default
```

Listing 4.20: *Adding the plugin on the fly.*

4.3.2 Generating the necessary resources

For the plugin to execute all of its functions properly, specific values need to be set. These are the following:

- Backup Target URL: [19] This is the endpoint of the Backupstore. Its form is `s3://<bucketname>@<region>/`. In our case, where the Backupstore is a MinIO server, we use `s3://backupbucket@us-east-1/`.
- Backup Target Secret: This is a Secret resource that the user needs to add to their Kubernetes cluster. It lives in `longhorn-system` namespace and contains all the

necessary information for the Backupstore including the ACCESS_KEY_ID, the SECRET_ACCESS_KEY, the ENDPOINT and whether it uses SSL. All of these values need to be added encoded in base64.

```
#minio-secret.yaml
apiVersion: v1
kind: Secret
metadata:
    name: minio-secret
    namespace: longhorn-system
type: Opaque
data:
    AWS_ACCESS_KEY_ID: <base64 encoded value>
    AWS_SECRET_ACCESS_KEY: <base64 encoded value>
    AWS_ENDPOINTS: <base64 encoded value>
    USESSL: <base64 encoded value>
```

Listing 4.21: *The secret of the Backup Target.*

- Lonhghorn Endpoint: This is the endpoint (IP address) of the Longhorn API server. In a Local Longhorn installation it can be the IP address of the master node.

- Destination Secret: This is a Secret that contains all the necessary information for the cluster where the volume data will be synced. It needs to include the same fields as the Backup Target Secret.

```
#minio-secret-destination.yaml
apiVersion: v1
kind: Secret
metadata:
    name: minio-secret-dst
    namespace: longhorn-system
type: Opaque
data:
    AWS_ACCESS_KEY_ID: <base64 encoded value>
    AWS_SECRET_ACCESS_KEY: <base64 encoded value>
    AWS_ENDPOINTS: <base64 encoded value>
    USESSL: <base64 encoded value>
```

Listing 4.22: *The secret of the destination bucket.*

If the user wants to hide the names of the resources, they have the option to create a file and use environmental variables. For example, in a file called `credentials` they can add the necessary values.

```
BACKUPTARGETURL=<Backupstore endpoint>
BACKUPTARGETSECRET=<name of secret>
LHENDPOINT=<Longhorn endpoint>
DSTSECRET=<name of secret>
```

Listing 4.23: *The credentials file.*

Then, they can use the `source credentials` command to set the environmental variables and use the name of the variable in the VolumeSnapshotLocation config.

To sum up, the users need to do the following to use the plugin in a cluster.

1. Create the necessary resources.
2. Create a credentials file.
3. Add the environmental variables.
4. Add the plugin (either by installing Velero from scratch or by applying the on the fly methid mentioned above).
5. Edit the VolumeSnapshotLocation to use the plugin as the snapshot provider.

4.3.3 Restoring from the backup bucket

After the installation of the plugin, every time that the user takes a backup of the cluster using Velero the volume data will not only go to the Backupstore, but will also be synced to another bucket (let's call this the `backup bucket`). This way, if the bucket used as the Backupstore becomes for any reason unavailable, the user can set the Backup Target to point to the backup bucket and all the backups will be present again.

To do this, the user can use Longhorn UI and navigate to `Setting` -> `General`. There they will find the setting called `Backup Target Credential Secret` and they can edit it to match the name of the secret added of the backup bucket.

If they navigate to the `Backup` page of the Longhorn UI, they will find all backups available in the backup bucket.

5 Evaluation

In this chapter, we present an evaluation of our design and implementation, by providing some metrics. To do so, we will examine all methods possible to create a backup using Velero.

5.1 Benchmark Creation

To be able to test all methods following the same procedures, we created a bash script that benchmarks Velero backups for different types (CSI, File System Backup and CSI Snapshot Data Movement), as well as different plugins (CSI plugin, our custom Velero plugin for Longhorn) by:

1. Setting up a test pod with a Longhorn Volume mounted.
2. Creating a user configured amount of data inside this volume.
3. Creating Velero backups at various stages (initial, modified files, new files), for various datasets and using different volume backup methods.
4. Recording timestamps to calculate backup duration for each stage and type.

Now, let's examine this script.

```
1  #! /bin/bash
2
3  usage() { echo "Usage: $0 [-s <filesize in kb>] [-b <backupname>] [-n <numberoffiles>] [-a <↩
       namespace>] [-t <backuptype>]" 1>&2; exit 1; }
4
5  while getopts "s:b:n:a:t:" flag
6  do
7          case "${flag}" in
8              s) size=${OPTARG};;
9              b) backupname=${OPTARG};;
10             n) numoffiles=${OPTARG};;
```

```
            a) namespace=${OPTARG};;
            t) backuptype=${OPTARG};;
            *) usage;;
        esac
done

if [ -z "$size" ] || [ -z "$backupname" ] || [ -z "$numoffiles" ] || [ -z "$namespace" ] || [ -z "$backuptype" ]; then
        usage
        exit 1
fi

if [ "$backuptype" == "fsb" ] || [ "$backuptype" == "csi" ] || [ "$backuptype" == "dm" ]; then
        echo "Executing backup: $backuptype"
else
        echo "Invalid argument. Please use csi, fsb or dm"
        exit 1
fi


kubectl apply -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
    name: $namespace
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: data-pvc
    namespace: $namespace
spec:
    storageClassName: longhorn-new
    accessModes:
    - ReadWriteOnce
    resources:
    requests:
        storage: 20Gi
---
apiVersion: v1
kind: Pod
metadata:
    name: testpod
    namespace: $namespace
spec:
    containers:
    - name: file-creator
    image: alpine
    command: ["/bin/sh", "-c", "echo numoffiles=$numoffiles >> ~/.bashrc; echo size=$size >> ~/.bashrc; tail -f /dev/null"]
    volumeMounts:
    - name: storage
        mountPath: /data
    volumes:
    - name: storage
    persistentVolumeClaim:
        claimName: data-pvc
EOF

kubectl wait --for=condition=ready pod -n $namespace testpod --timeout=300s
kubectl exec -it -n $namespace testpod -- /bin/sh -c "'cat create-files.sh'"

if [ "$backuptype" == "csi" ]; then
        velero backup create --include-namespaces $namespace $backupname-initial --wait
        velero backup describe $backupname-initial
elif  [ "$backuptype" == "fsb" ]; then
        velero backup create --include-namespaces $namespace --default-volumes-to-fs-backup $backupname-initial --wait
        velero backup describe $backupname-initial
elif  [ "$backuptype" == "dm" ]; then
        velero backup create --include-namespaces $namespace --snapshot-move-data $backupname-initial --wait
        velero backup describe $backupname-initial
else
        echo "You have not selected a valid backup type!"
fi

now=$(date +%T)
filename=$backuptype-$namespace-$now
velero backup describe $backupname-initial > $filename
bckpini="${backupname}-initial"
started_line=$(velero backup describe ${bckpini} | grep "^Started:")
completed_line=$(velero backup describe ${bckpini} | grep "^Completed:")

# Extract timestamps from the lines
started_timestamp=$(echo "$started_line" | awk '{print $2, $3}')
completed_timestamp=$(echo "$completed_line" | awk '{print $2, $3}')

# Convert timestamps to seconds since epoch
started_seconds=$(date -d "$started_timestamp" +%s)
completed_seconds=$(date -d "$completed_timestamp" +%s)

# Calculate the difference in seconds
```

```
difference_seconds=$((completed_seconds - started_seconds))

# Convert the difference to a human-readable format
difference=$(date -u -d @${difference_seconds} +"%H:%M:%S")

echo "---" >> $filename
echo "Started: $started_timestamp" >> $filename
echo "Completed: $completed_timestamp" >> $filename
echo "Duration: $difference" >> $filename

echo "Touching files!"
kubectl exec -it -n $namespace testpod -- /bin/sh -c "num=$numoffiles; 'cat touch-files.sh'"
echo "Touching files completed!"

if [ "$backuptype" == "csi" ]; then
        velero backup create --include-namespaces $namespace $backupname-touch --wait
        velero backup describe $backupname-touch
elif  [ "$backuptype" == "fsb" ]; then
        velero backup create --include-namespaces $namespace --default-volumes-to-fs-backup ↩
            $backupname-touch --wait
        velero backup describe $backupname-touch
elif  [ "$backuptype" == "dm" ]; then
        velero backup create --include-namespaces $namespace --snapshot-move-data $backupname-↩
            touch --wait
        velero backup describe $backupname-touch
else
        echo "You have not selected a valid backup type!"
fi

bckptouch="${backupname}-touch"
started_line=$(velero backup describe ${bckptouch} | grep "^Started:")
completed_line=$(velero backup describe ${bckptouch} | grep "^Completed:")

# Extract timestamps from the lines
started_timestamp=$(echo "$started_line" | awk '{print $2, $3}')
completed_timestamp=$(echo "$completed_line" | awk '{print $2, $3}')

# Convert timestamps to seconds since epoch
started_seconds=$(date -d "$started_timestamp" +%s)
completed_seconds=$(date -d "$completed_timestamp" +%s)

# Calculate the difference in seconds
difference_seconds=$((completed_seconds - started_seconds))

# Convert the difference to a human-readable format
difference=$(date -u -d @${difference_seconds} +"%H:%M:%S")

now=$(date +%T)
echo "---" >> $filename
velero backup describe $backupname-touch >> $filename
echo "---" >> $filename
echo "Started: $started_timestamp" >> $filename
echo "Completed: $completed_timestamp" >> $filename
echo "Duration: $difference" >> $filename

echo "Adding inc files!"
kubectl exec -it -n $namespace testpod -- /bin/sh -c "'cat inc-files.sh'"
echo "Adding inc files completed!"

if [ "$backuptype" == "csi" ]; then
        velero backup create --include-namespaces $namespace $backupname-inc --wait
        velero backup describe $backupname-inc
elif  [ "$backuptype" == "fsb" ]; then
        velero backup create --include-namespaces $namespace --default-volumes-to-fs-backup ↩
            $backupname-inc --wait
        velero backup describe $backupname-inc
elif  [ "$backuptype" == "dm" ]; then
        velero backup create --include-namespaces $namespace --snapshot-move-data $backupname-↩
            inc --wait
        velero backup describe $backupname-inc
else
        echo "You have not selected a valid backup type!"
fi

bckpinc="${backupname}-inc"
started_line=$(velero backup describe ${bckpinc} | grep "^Started:")
completed_line=$(velero backup describe ${bckpinc} | grep "^Completed:")

# Extract timestamps from the lines
started_timestamp=$(echo "$started_line" | awk '{print $2, $3}')
completed_timestamp=$(echo "$completed_line" | awk '{print $2, $3}')

# Convert timestamps to seconds since epoch
started_seconds=$(date -d "$started_timestamp" +%s)
completed_seconds=$(date -d "$completed_timestamp" +%s)

# Calculate the difference in seconds
difference_seconds=$((completed_seconds - started_seconds))

# Convert the difference to a human-readable format
difference=$(date -u -d @${difference_seconds} +"%H:%M:%S")

```

```
188 now=$(date +%T)
189 echo "---" >> $filename
190 velero backup describe $backupname-inc >> $filename
191 echo "---" >> $filename
192 echo "Started: $started_timestamp" >> $filename
193 echo "Completed: $completed_timestamp" >> $filename
194 echo "Duration: $difference" >> $filename
195
196 echo "Benchmark process completed!"
```

Listing 5.1: *The benchmark generation script.*

Here is the sequence of the script. It performs the following:

1. Define its usage.
2. Perform validation tests to ensure proper usage.
3. Create three resources in the cluster: a namespace, a PersistentVolumeClaim, and a pod that uses this PersistentVolumeClaim.
4. Deploy a sub-script to generate the requested random files inside the new Longhorn Volume.
5. Execute the requested backup.
6. Add the output of the velero describe command for this backup to a file including benchmarks for this script run.
7. Add the backup timestamps and the total duration of the backup in the file.
8. Perform a touch operation on all the created files using a sub-script.
9. Take an additional backup and add the corresponding output to the benchmarks file.
10. Create 30000 additional files in the volume.
11. Take one more backup and add the corresponding output to the benchmarks file.
12. Delete the created resources.

As mentioned above, this script utilizes some sub-scripts, mainly to generate the wanted files inside the Longhorn Volume. These sub-scripts are the following:

- **create-files.sh**: Create random files inside the volume, the number and size of which are specified by the user at script startup time.

```
1 i=1;
2
3 while [ $i -le $numoffiles ]; do
4         dd if=/dev/urandom of=/data/file$( printf %03d "$i" ).bin bs=1K count=$size > /dev/↩
              null 2>&1
5         echo "File created with number: " $i
6         i=$((i + 1))
7 done
```

Listing 5.2: *The file generation sub-script.*

- **touch-files.sh**: Perform a `touch` operation on all the previously created files. This tests the backup behavior after the files have been accessed (we change their timestamps).

```
i=1;

while [ $i -le $num ]; do
        touch /data/file$( printf %03d "$i" ).bin
        echo "Touched file with number: " $i
        i=$((i + 1))
done
```

Listing 5.3: *The file touching sub-script.*

- **inc-files.sh**: Add 30000 new files with a size of 1KB. This has been arbitrarily decided, to demonstrate the backup capabilities after some new files have been created.

```
i=1;

while [ $i -le 30000 ]; do
        dd if=/dev/urandom of=/data/add_file$( printf %03d "$i" ).bin bs=1K count=10 > /dev/↩
             null 2>&1
        echo "File created with number: " $i
        i=$((i + 1))
done
```

Listing 5.4: *The additional files generation sub-script.*

When trying to test and create benchmarks for the File System Backup of Velero (using kopia as the provider) or when trying to test Velero Built-In Data Mover (which also uses kopia), I came across the following bug.

```
time="timestamp" level=info msg="Searching for parent snapshot" ...
panic: counter cannot decrease in value
```

This is a `kopia` error, which has to do with the timing of the files. [77] Intel and AMD have created processor extensions to the x86 architecture (Intel VT-x and AMD-V). In the VirtualBox logs I could see the following message:

```
{timestamp} HM: HMR3Init: Attempting fall back to NEM: AMD-V is not available
{timestamp} NEM: WHvCapabilityCodeHypervisorPresent is TRUE, so this might work...
```

while AMD-V was activated in my CPU. Another sign that there was something wrong with the virtualization of Virtualbox was that there was present a green turtle icon as displayed in Figure 5.1. This means that Hyper-V is active on my PC and this is interfering with other programs that need Virtualization Technology. To solve this, we disabled Hyper-V at the Windows host:

```
bcdedit /set hypervisorlaunchtype off
```

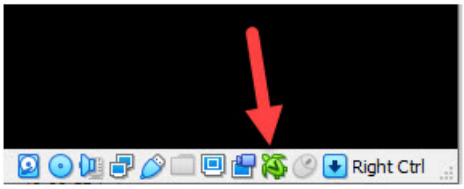


Figure 5.1: *The icon VirtualBox displays.*

5.2 Results

To test the snapshotting mechanisms in various conditions, we will define three datasets each of which will consist of about 10GB of data.

- The **large-file** dataset will include 1 file with 10GB size.
- The **medium-file** dataset will include 1000 files with 10MB size each.
- The **small-file** dataset will include 625000 files with 16KB each.

We tested these datasets that contain random data against the described backup mechanisms: CSI Snapshot using the common CSI plugin, CSI Snapshot using Velero Plugin for Longhorn, File System Backup (with `kopia` as the provider) and Velero CSI Snapshot Data Movement. The CSI snapshot using the CSI plugin was set to create Longhorn Backups. We present the results below for each test. For each snapshot type, we have three rows regarding the time it took to execute the backup: one for the initial backup (called `initial`), one for the backup after touching all created files (called `touch`) and one after adding the 30000 new files (called `inc`).

5.2.1 CSI Snapshot using CSI plugin

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
CSI	Initial	1	10000000	2min 7 sec
CSI	After touch	1	10000000	20 sec
CSI	After inc	1	10000000	18 sec

As expected, the initial backup is the one that takes the most time. Since Longhorn's CSI snapshot mechanism uses Longhorn Backups (which is a block based backup method), it is expected that the additional backup operations will function incrementally on the first and thus will require significantly less time.

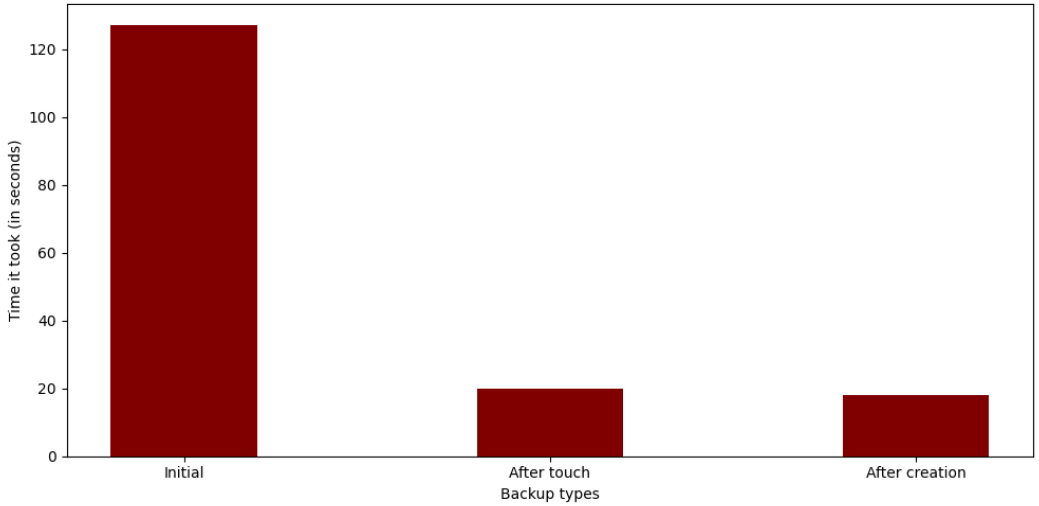


Figure 5.2: *The benchmarks with CSI plugin for 1 file of 10GB.*

Then we have the benchmarks for CSI plugin when using the medium dataset.

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
CSI	Initial	1000	10000	2min 5 sec
CSI	After touch	1000	10000	15 sec
CSI	After inc	1000	10000	21 sec

We can see that the results are almost identical to the large dataset.

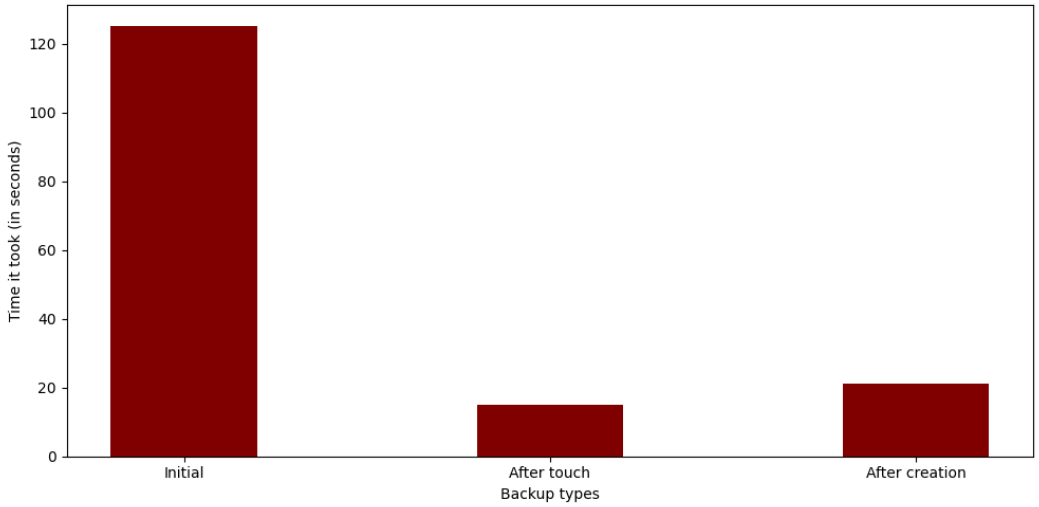


Figure 5.3: *The benchmarks with CSI plugin for 1000 files of 10MB.*

Finally, we have the small dataset.

The results continue being very similar to the other datasets, confirming that since CSI Snapshot is a block based method, it is indeed very fast.

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
CSI	Initial	625000	16	1min 40 sec
CSI	After touch	625000	16	19 sec
CSI	After inc	625000	16	29 sec

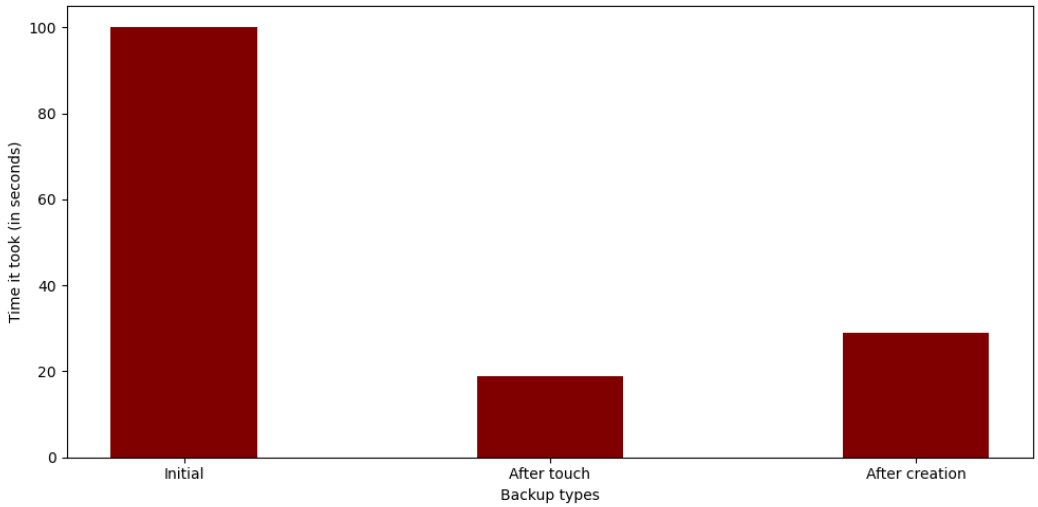


Figure 5.4: *The benchmarks with CSI plugin for 625000 files of 16KB.*

Examining the above results, we can see that with block based backup methods, it makes no big difference when it comes to backup time. The incremental nature of these backups is also really useful, whether it comes to backing up a touched file or additional files.

5.2.2 File System Backup with Kopia provider

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
FSB	Initial	1	10000000	2min 51 sec
FSB	After touch	1	10000000	1min 18 sec
FSB	After inc	1	10000000	10 sec

When examining the File System Backup, this is where we get our first differences. While this method is somewhat slower than CSI Snapshot during the initial snapshotting, the biggest difference is the time it took to complete the snapshot after performing the touch operation on all files. File System Backup could not efficiently skip the backup files that had already been backed up, resulting to a much slower backup operation. The backup after adding the extra files, was equally fast with the CSI Snapshot.

Below, we can see the results of the medium dataset.

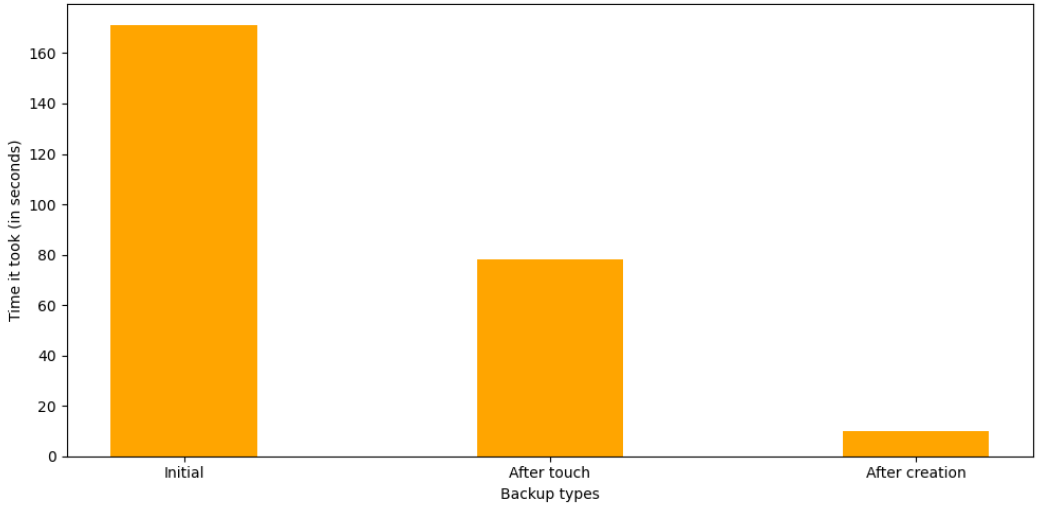


Figure 5.5: *The benchmarks with File System Backup for 1 file of 10GB.*

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
FSB	Initial	1000	10000	2min 28 sec
FSB	After touch	1000	10000	1min 4 sec
FSB	After inc	1000	10000	13 sec

The results resemble the ones on the large dataset significantly.

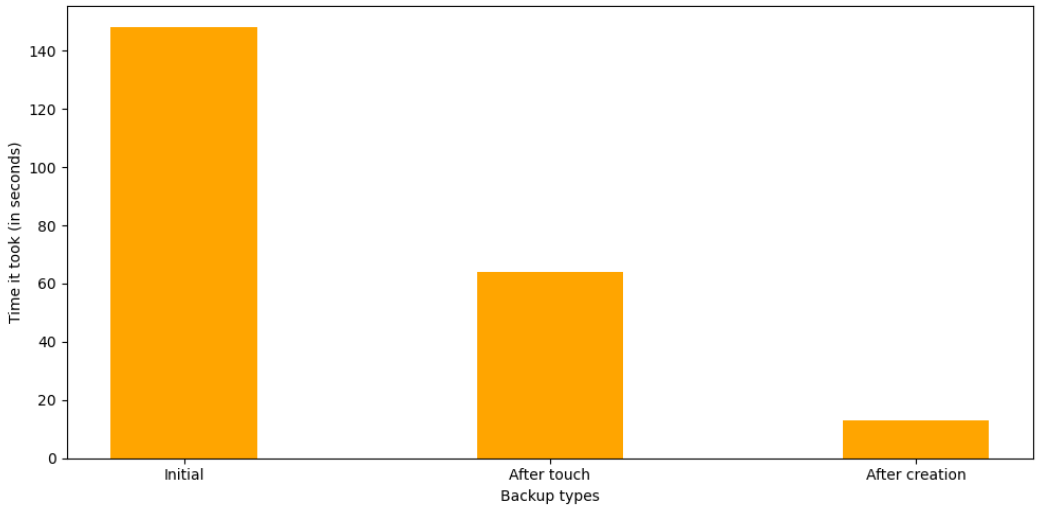


Figure 5.6: *The benchmarks with File System Backup for 1000 file of 10MB.*

Finally, we have the results for the small dataset.

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
FSB	Initial	625000	16	9min 3 sec
FSB	After touch	625000	16	7min 36 sec
FSB	After inc	625000	16	25 sec

Here comes a significant difference compared the other datasets. File System Backup takes much more time to back up the data, when multiple small files are included. This comes as a contrast to the CSI Snapshot, where regardless of the number and size of files included in the volume, the backup time was similar. The subsequent backups for the large dataset follow the same pattern as the other datasets, as it takes much time to perform the backup after a touch operation and smaller amount of time to perform the backup with the additional files.

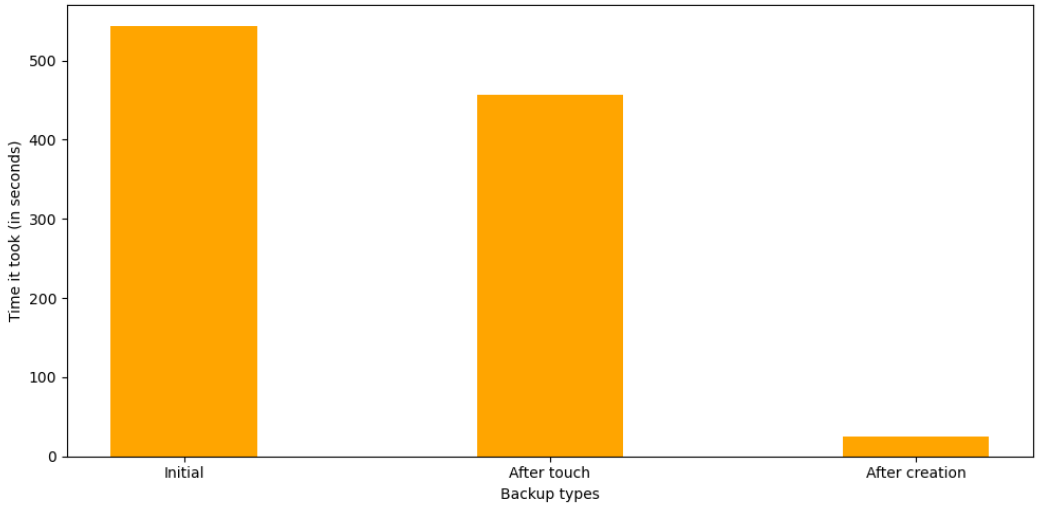


Figure 5.7: *The benchmarks with File System Backup for 625000 file of 16KB.*

5.2.3 CSI Snapshot Data Movement

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
DM	Initial	1	10000000	6min 44sec
DM	After touch	1	10000000	4min 2sec
DM	After inc	1	10000000	2min 27sec

As we can see, CSI Snapshot Data Movement takes significantly more time than the other techniques. This is to be expected, as it is essentially a combination of the other two methods. Here we can see that the initial backup takes more time than the others, while the backup after touching the file also takes more time than the third one. Presumably, this could be due to the involvement of File System Backup.

Then, we have the results for the medium dataset.

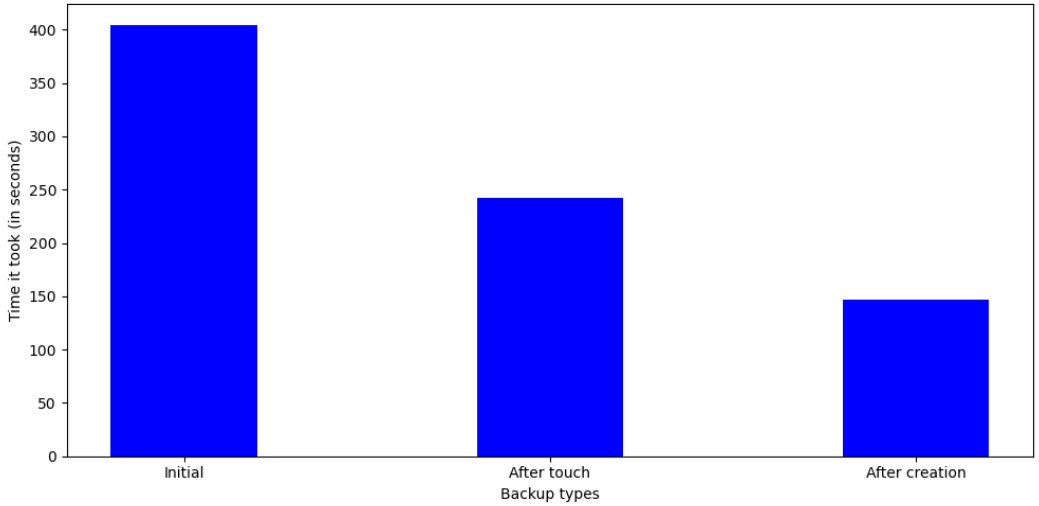


Figure 5.8: *The benchmarks with CSI Snapshot Data Movement for 1 file of 10GB.*

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
DM	Initial	1000	10000	7min 10sec
DM	After touch	1000	10000	6min 28sec
DM	After inc	1000	10000	4min 37sec

The results here are similar to the large backup dataset, but the times are somewhat larger. This could be explained by the fact that File System Backup is slower with more files.

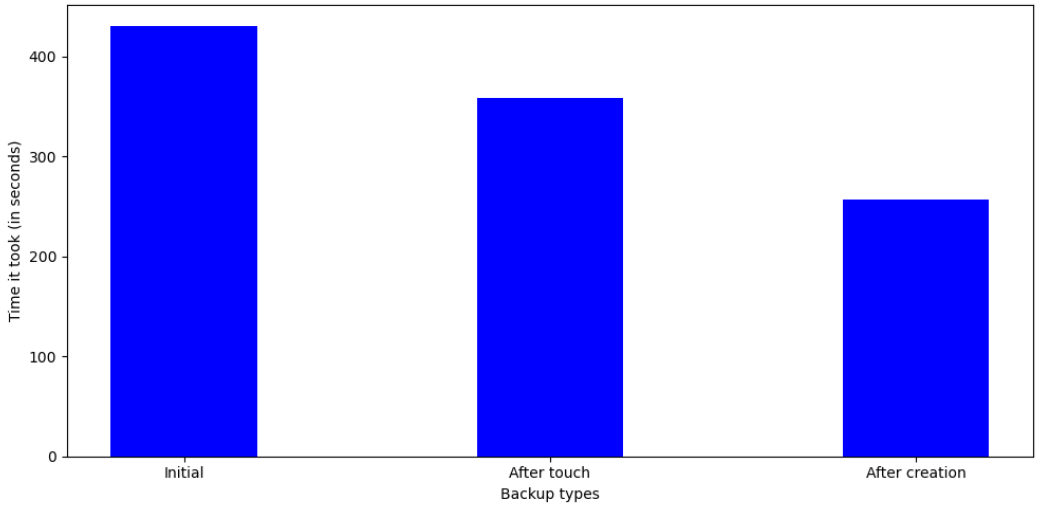


Figure 5.9: *The benchmarks with CSI Snapshot Data Movement for 1000 files of 10MB.*

Finally we have the small dataset.

What we have seen so far is also confirmed here. CSI Snapshot Data Movement is much

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
DM	Initial	625000	16	15min 29sec
DM	After touch	625000	16	11min 59sec
DM	After inc	625000	16	5min 27sec

slower than both of the other methods, and this gets worse as we increase the number of the files. The two subsequent backups follow the trend of the other datasets.

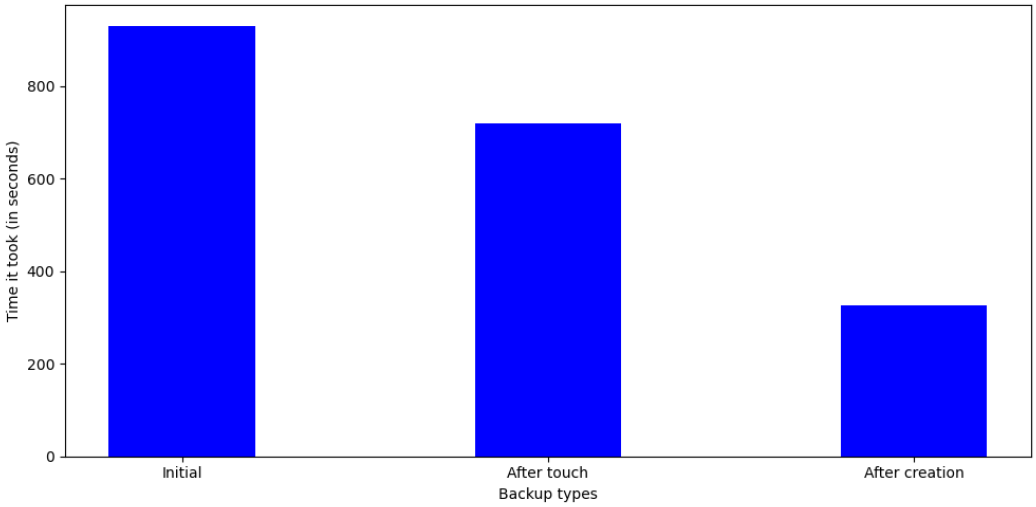


Figure 5.10: *The benchmarks with CSI Snapshot Data Movement for 625000 files of 16KB.*

5.2.4 CSI Snapshot using Velero Plugin for Longhorn

Snapshot Type	Stage	No. of Files	File size (in kB)	Time it took
CSI (Longhorn Plugin)	Initial	1	10000000	4min 25sec
CSI (Longhorn Plugin)	After touch	1	10000000	2min 35sec
CSI (Longhorn Plugin)	After inc	1	10000000	2min 39sec

Since the most logical comparison here would be with CSI Snapshots using the CSI plugin, we will focus on this. We can see that the backups using the Velero Plugin for Longhorn are generally slower than when using the generic CSI plugin. This makes sense, as in this case, we trigger a Longhorn Backup using the Longhorn API, wait until the backup is ready, and then sync all the files from the Backupstore to the destination bucket. This is why all backups are slower when using Velero Plugin for Longhorn. This

also means that the larger the backed up volume is, the bigger the delay will be when using our custom plugin. Also, just like the CSI plugin, both the `touch` and the `inc` backups take similar time to complete, as expected in a block based backup.

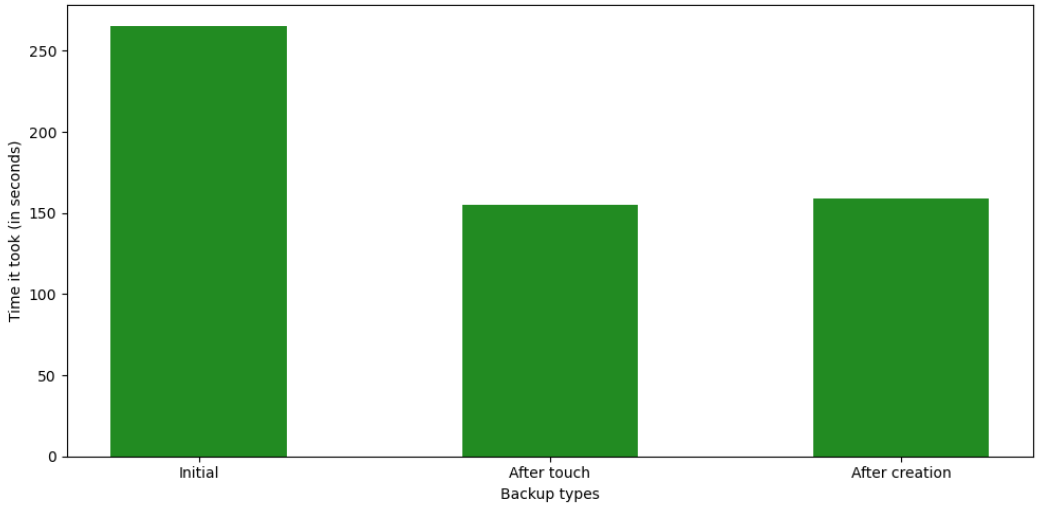


Figure 5.11: *The benchmarks with Velero Plugin for Longhorn for 1 file of 10GB.*

Then, we have the results for the medium dataset.

Snapshot Type	Stage	Number of Files	File size (in kB)	Time it took
CSI (Longhorn Plugin)	Initial	1000000	10	4min 41sec
CSI (Longhorn Plugin)	After touch	1000000	10	2min 47sec
CSI (Longhorn Plugin)	After inc	1000000	10	2min 45sec

As expected, the results are nearly identical to the large dataset. Note that we also observed the same behavior with the CSI plugin.

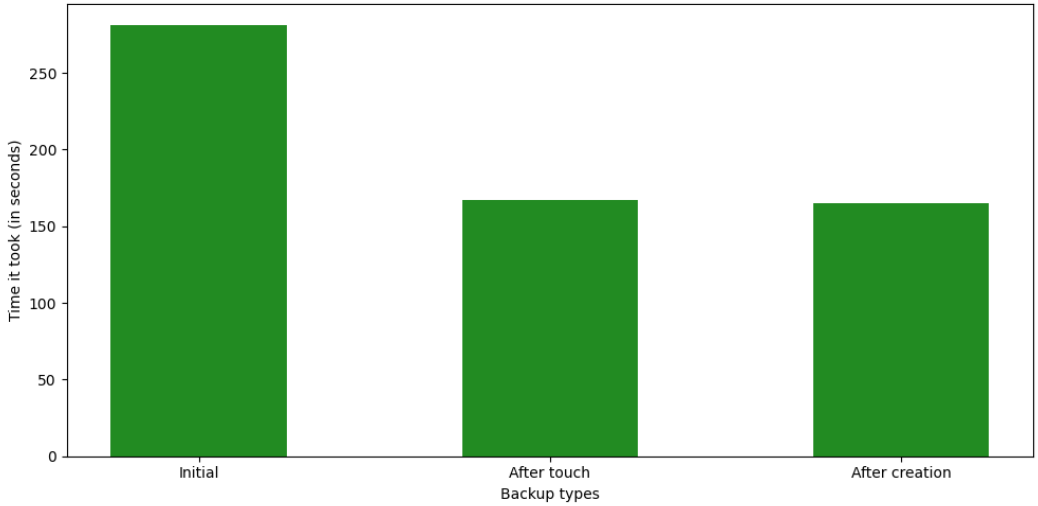


Figure 5.12: *The benchmarks with Velero Plugin for Longhorn for 1000 files of 10MB.*

Finally, we have the small dataset.

Snapshot Type	Stage	Number of Files	File size (in kB)	Time it took
CSI (Longhorn Plugin)	Initial	625000	16	4min 5sec
CSI (Longhorn Plugin)	After touch	625000	16	2min 33sec
CSI (Longhorn Plugin)	After inc	625000	16	2min 37sec

We can see the same behavior, where the distribution of data in a volume does not affect the block based backup regarding the time it takes to complete.

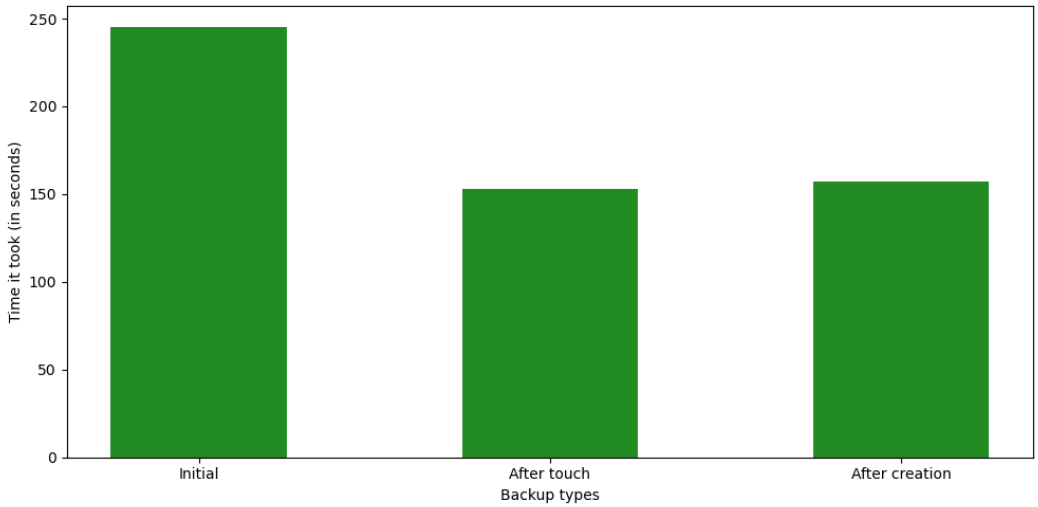


Figure 5.13: *The benchmarks with Velero Plugin for Longhorn for 625000 file of 16KB.*

5.2.5 Remarks

Examining the benchmarks analyzed above, we can come to some interesting observations.

- The CSI snapshot is generally faster than compared to FSB and DM, as expected.
- FSB is slower than CSI but faster than DM. This makes sense, as DM involves the process of both taking a CSI snapshot and after creating a new volume from it, performing FSB on the new volume.
- DM is slower than both of these methods, but gives a more stable result than FSB which performs the backup operation on the live file system.
- The CSI snapshotting mechanism has a similar performance on both the case of modifying (`touch`) the existing files and adding extra (`inc`) files.
- Both the mechanisms that include FSB (FSB and DM) perform remarkably better on new files compared to modified files (even if the only thing changed is the timestamp of the file).
- Velero Plugin for Longhorn is generally slower than CSI plugin, which makes sense as it performs the same backup operation and then syncs the data to an additional bucket.

6 Conclusion

We are now approaching the end of this thesis. In this chapter, we will try to outline the journey that led to its completion and propose future steps that could step on this foundation and expand it.

6.1 Concluding Remarks

We started this process by exploring Kubernetes and its capabilities, finding ways to deploy and test clusters locally, and figuring out the ins and outs of Velero. When Longhorn came into play, the goal became clearer: develop an easy and automated way to integrate Velero's capabilities with a storage system that does not have a native way of doing this. However, this was not enough since Longhorn supports CSI, and Velero provides us with a CSI plugin. We strived to add an extra capability that would make sense for every system following our architecture. Since Longhorn does not allow the creation of multiple Backup Targets, the challenge became obvious: having the ability to store the backup data in a secondary location and restore from it if needed. On top of all that, the goal was clear: make everything as automated as possible and work like a breeze.

When the goal was set, the path included studying the Velero plugin mechanism, various SDKs for Go (including Longhorn and MinIO), and multiple attempts to solve the above problem. Our solution involved syncing all objects at the time of each backup so that the secondary storage location has contents identical to the primary one we used. And if we need to perform a restore operation from the secondary storage location, the restore

process needs the user only to set the secondary storage as the Backup Target and execute the backup normally.

Our benchmark results showed the comparison between the existing backup and restore solutions and how the Velero plugin for Longhorn stands against them. They also confirmed the grade at which it accomplishes its targets.

6.2 Future Work

Concluding this thesis, it became apparent that we achieved our initial target: making syncing volume data to a secondary location when using Velero with Longhorn easier. However, we could expand the work done on this project in the following ways.

- Expand the synchronizing operation not only on the backup volume data but also on the manifests. Velero executes the backup and then sends the backup data to the Backup Storage Location set by the user. We could also sync this data to the secondary location the user has set.
- Extend the plugin to handle setups using different types of Backupstores besides S3, like GCP Cloud Storage, NFS, and Azure Blob Storage.
- Contribute and push the plugin to Velero so that Longhorn becomes a part of the Velero-supported providers. [20]

Bibliography

[1] K3s, "Lightweight kubernetes," https://k3s.io/, (Accessed Nov. 1, 2024).

[2] Veeam, "The 3-2-1 backup rule," https://www.veeam.com/blog/321-backup-rule.html, October 2022, (Accessed Nov. 1, 2024).

[3] TechTarget, "What is off-site backup?" https://www.techtarget.com/searchdatabackup/definition/off-site-backup, May 2023, (Accessed Nov. 1, 2024).

[4] Velero, "File system backup," https://velero.io/docs/v1.14/file-system-backup/, 2024, (Accessed Nov. 1, 2024).

[5] VMware, "What is a virtual machine?" https://www.vmware.com/topics/virtual-machine, (Accessed Nov. 1, 2024).

[6] ScienceDirect, "Process virtual machine," https://www.sciencedirect.com/topics/computer-science/process-virtual-machine, (Accessed Nov. 1, 2024).

[7] G. Cloud, "What is container orchestration?" https://cloud.google.com/discover/what-is-container-orchestration, (Accessed Nov. 1, 2024).

[8] IBM, "What is kubernetes?" https://www.ibm.com/topics/kubernetes, (Accessed Nov. 1, 2024).

[9] Akamai, "What is managed kubernetes?" https://www.akamai.com/glossary/what-is-managed-kubernetes, (Accessed Nov. 1, 2024).

[10] Kubernetes, “Concepts - overview,” https://kubernetes.io/docs/concepts/overview/, 2024, (Accessed Nov. 1, 2024).

[11] K3s, “K3s documentation,” https://docs.k3s.io/, (Accessed Nov. 1, 2024).

[12] Longhorn, “Longhorn documentation,” https://longhorn.io/docs/, (Accessed Nov. 1, 2024).

[13] Longhorn, “Concepts - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/concepts/, 2024, (Accessed Nov. 1, 2024).

[14] DigitalOcean, “How to set up minio object storage server in standalone mode on ubuntu 20.04,” https://www.digitalocean.com/community/tutorials/how-to-set-up-minio-object-storage-server-in-standalone-mode-on-ubuntu-20-04, 2020, (Accessed Nov. 1, 2024).

[15] Velero, “Velero documentation v1.14,” https://velero.io/docs/v1.14/, (Accessed Nov. 1, 2024).

[16] K. CSI, “Kubernetes container storage interface (csi) documentation,” https://kubernetes-csi.github.io/docs/, (Accessed Nov. 1, 2024).

[17] Velero, “Csi snapshot data movement - velero documentation,” https://velero.io/docs/main/csi-snapshot-data-movement/, (Accessed Nov. 1, 2024).

[18] Velero, “Velero volume snapshotter - volume_snapshotter.go,” https://github.com/vmware-tanzu/velero/blob/main/pkg/plugin/velero/volumesnapshotter/v1/volume_snapshotter.go, (Accessed Nov. 1, 2024).

[19] Longhorn, “Set up a local testing backupstore - longhorn documentation (v1.7.2),” https://longhorn.io/docs/1.7.2/snapshots-and-backups/backup-and-restore/set-backup-target/#set-up-a-local-testing-backupstore, (Accessed Nov. 1, 2024).

[20] Velero, “Supported providers - velero documentation,” https://velero.io/docs/main/supported-providers/, (Accessed Nov. 1, 2024).

[21] TechTarget, “What is the 3-2-1 backup strategy?” https://www.techtarget.com/searchdatabackup/definition/3-2-1-Backup-Strategy, April 2023, (Accessed Nov. 1, 2024).

[22] V. Forum, “Discussing: Windows 10 as a guest - tips and tricks,” https://forums.virtualbox.org/viewtopic.php?f=25&t=99390, 2020, (Accessed Nov. 1, 2024).

[23] Oracle, “Hyper-v support,” https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/hyperv-support.html, 2020, (Accessed Nov. 1, 2024).

[24] M. Kaschke, “Virtual machine (vm) vs container,” https://mkaschke.medium.com/virtual-machine-vm-vs-container-13ab51f4c177, 2021, (Accessed Nov. 1, 2024).

[25] Kubernetes, “Kubernetes components overview - kubernetes documentation,” Official Kubernetes Documentation, (Accessed Nov. 1, 2024). [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[26] Kubernetes, “Volumes - kubernetes documentation,” https://kubernetes.io/docs/concepts/storage/volumes/, (Accessed Nov. 1, 2024).

[27] Rancher, “Managing persistent storage,” https://ranchermanager.docs.rancher.com/how-to-guides/new-user-guides/manage-clusters/create-kubernetes-persistent-storage/manage-persistent-storage/about-persistent-storage/, (Accessed Nov. 1, 2024).

[28] K. CSI, “Kubernetes container storage interface (csi) documentation,” https://kubernetes-csi.github.io/docs/#kubernetes-container-storage-interface-csi-documentation, (Accessed Nov. 1, 2024).

[29] Kubernetes, “Container storage interface (csi) ga,” https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/, January 2019, (Accessed Nov. 1, 2024).

[30] K. CSI, “Sidecar containers,” https://kubernetes-csi.github.io/docs/sidecar-containers.html, (Accessed Nov. 1, 2024).

[31] K. CSI, “Cluster driver registrar,” https://kubernetes-csi.github.io/docs/cluster-driver-registrar.html, (Accessed Nov. 1, 2024).

[32] Kubernetes, “Container storage interface (csi) design proposal,” https://github.com/kubernetes/design-proposals-archive/blob/main/storage/container-storage-interface.md, (Accessed Nov. 1, 2024).

[33] Kubernetes, “Volume snapshots,” https://kubernetes.io/docs/concepts/storage/volume-snapshots/, 2024, (Accessed Nov. 1, 2024).

[34] Kubernetes, “Volume snapshot classes,” https://kubernetes.io/docs/concepts/storage/volume-snapshot-classes/, 2024, (Accessed Nov. 1, 2024).

[35] SUSE, “Introduction to k3s,” https://www.suse.com/c/rancher_blog/introduction-to-k3s/, 2020, (Accessed Nov. 1, 2024).

[36] K. Team, “Kine - a lightweight alternative to etcd for k3s,” https://github.com/k3s-io/kine, (Accessed Nov. 1, 2024).

[37] K3s, “K3s architecture,” https://docs.k3s.io/architecture, (Accessed Nov. 1, 2024).

[38] K3s, “K3s storage,” https://docs.k3s.io/storage, (Accessed Nov. 1, 2024).

[39] Rancher, “Local path provisioner,” https://github.com/rancher/local-path-provisioner/, (Accessed Nov. 1, 2024).

[40] Longhorn, “Longhorn volume - terminology (v1.6.2),” https://longhorn.io/docs/1.6.2/terminology/#longhorn-volume, 2024, (Accessed Nov. 1, 2024).

[41] Longhorn, “Longhorn engine,” https://github.com/longhorn/longhorn-engine, (Accessed Nov. 1, 2024).

[42] K. CSI, “External provisioner - readme,” https://github.com/kubernetes-csi/external-provisioner/blob/46212fd905099726440823e6e1c8d22f666b232f/README.md, (Accessed Nov. 1, 2024).

[43] Longhorn, “Issue #5315 - github,” https://github.com/longhorn/longhorn/issues/5315, (Accessed Nov. 1, 2024).

[44] MinIO, “Erasure coding - minio documentation,” https://min.io/docs/minio/linux/operations/concepts/erasure-coding.html, (Accessed Nov. 1, 2024).

[45] Google, “Highwayhash,” https://github.com/google/highwayhash, (Accessed Nov. 1, 2024).

[46] MinIO, “Minio product overview,” https://min.io/product/overview, (Accessed Nov. 1, 2024).

[47] MinIO, “Data authenticity and integrity - minio blog,” https://blog.min.io/data-authenticity-integrity/, (Accessed Nov. 1, 2024).

[48] MinIO, “Architecture concepts - minio documentation for windows,” https://min.io/docs/minio/windows/operations/concepts/architecture.html, (Accessed Nov. 1, 2024).

[49] Kubernetes, “Controller - kubernetes architecture,” https://kubernetes.io/docs/concepts/architecture/controller/, (Accessed Nov. 1, 2024).

[50] Velero, “How velero works - velero documentation v1.14,” https://velero.io/docs/v1.14/how-velero-works/, (Accessed Nov. 1, 2024).

[51] Longhorn, “Snapshots - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/concepts/#24-snapshots, (Accessed Nov. 1, 2024).

[52] Longhorn, “Replicas - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/concepts/#23-replicas, (Accessed Nov. 1, 2024).

[53] Longhorn, “How read and write operations work for replicas - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/concepts/#231-how-read-and-write-operations-work-for-replicas, (Accessed Nov. 1, 2024).

[54] Longhorn, “How backups work - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/concepts/#31-how-backups-work, (Accessed Nov. 1, 2024).

[55] Longhorn, “Backup target - longhorn terminology (v1.6.2),” https://longhorn.io/docs/1.6.2/terminology/#backup-target, (Accessed Nov. 1, 2024).

[56] Longhorn, “Longhorn backup creation diagram,” https://longhorn.io/img/diagrams/concepts/longhorn-backup-creation.png, (Accessed Nov. 1, 2024).

[57] Longhorn, “Backing image - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/advanced-resources/backing-image/backing-image/, (Accessed Nov. 1, 2024).

[58] Longhorn, “Backing image backup - longhorn documentation (v1.6.2),” https://longhorn.io/docs/1.6.2/advanced-resources/backing-image/backing-image-backup/, (Accessed Nov. 1, 2024).

[59] Longhorn, "The checksum of a backing image - longhorn documentation (v1.6.2)," https://longhorn.io/docs/1.6.2/advanced-resources/backing-image/backing-image/#the-checksum-of-a-backing-image, (Accessed Nov. 1, 2024).

[60] Longhorn, "Csi plugin - longhorn documentation (v1.6.2)," https://longhorn.io/docs/1.6.2/concepts/#14-csi-plugin, (Accessed Nov. 1, 2024).

[61] Longhorn, "Csi snapshot support - longhorn enhancement proposal," https://github.com/longhorn/longhorn/blob/master/enhancements/20200904-csi-snapshot-support.md, (Accessed Nov. 1, 2024).

[62] Restic, "Restic: A fast, secure, and efficient backup program," https://restic.net/, (Accessed Nov. 1, 2024).

[63] Velero, "Unified repository and kopia integration - velero design document (v1.14.0)," https://github.com/vmware-tanzu/velero/blob/v1.14.0/design/Implemented/unified-repo-and-kopia-integration/unified-repo-and-kopia-integration.md, (Accessed Nov. 1, 2024).

[64] Kopia, "Kopia: Fast, secure, and encrypted backup," https://kopia.io/docs/, (Accessed Nov. 1, 2024).

[65] Kopia, "Features - kopia documentation," https://kopia.io/docs/features/, (Accessed Nov. 1, 2024).

[66] Kasten, "Benchmarking kopia: Architecture, scale, and performance," https://web.archive.org/web/20231202012341/https://www.kasten.io/kubernetes/resources/blog/benchmarking-kopia-architecture-scale-and-performance, (Accessed Nov. 1, 2024).

[67] Kopia, "Architecture - kopia documentation," https://kopia.io/docs/advanced/architecture/, (Accessed Nov. 1, 2024).

[68] Kopia, "Binary large object (blob) storage - kopia documentation," https://kopia.io/docs/advanced/architecture/#binary-large-object-storage-blob, (Accessed Nov. 1, 2024).

[69] Kopia, "Content-addressable object storage (caos) - kopia documentation," https://kopia.io/docs/advanced/architecture/#content-addressable-object-storage-caos, (Accessed Nov. 1, 2024).

[70] Kopia, "Label-addressable manifest storage (lams) - kopia documentation," https://kopia.io/docs/advanced/architecture/#label-addressable-manifest-storage-lams, (Accessed Nov. 1, 2024).

[71] Velero, "Volume snapshot data movement," GitHub, (Accessed Nov. 1, 2024). [Online]. Available: https://github.com/vmware-tanzu/velero/blob/v1.14.0/design/volume-snapshot-data-movement/volume-snapshot-data-movement.md

[72] Velero, "Restore order - velero documentation (v1.15)," https://velero.io/docs/v1.15/restore-reference/#restore-order, (Accessed Nov. 1, 2024).

[73] Velero, "Snapshot pv restore - velero documentation (v1.15)," https://velero.io/docs/v1.15/restore-reference/#snapshot-pv-restore, (Accessed Nov. 1, 2024).

[74] Velero, "Restore - file system backup - velero documentation (v1.15)," https://velero.io/docs/v1.15/file-system-backup/#restore, (Accessed Nov. 1, 2024).

[75] HashiCorp, "Go plugin - hashicorp github repository," https://github.com/hashicorp/go-plugin, (Accessed Nov. 1, 2024).

[76] Velero, "Velero plugin framework - server.go," https://github.com/vmware-tanzu/velero/blob/main/pkg/plugin/framework/server.go, (Accessed Nov. 1, 2024).

[77] Kopia, "Snapshot create panic: counter cannot decrease value · issue #3342 · kopia/kopia," https://github.com/kopia/kopia/issues/3342, September 2023, (Accessed November 1, 2024).