ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επέκταση του Συστήματος Διαχείρισης Εικονικών Μηχανών Google Ganeti για την Παροχή Υψηλής Διαθεσιμότητας με Αυτόματη Μετάπτωση του Κύριου Κόμβου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Α. Παρασύρης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2017

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

Επέκταση του Συστήματος Διαχείρισης Εικονικών Μηχανών Google Ganeti για την Παροχή Υψηλής Διαθεσιμότητας με Αυτόματη Μετάπτωση του Κύριου Κόμβου

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Α. Παρασύρης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου 2017.

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

Αθήνα, Ιούλιος 2017

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

Extending the Google Ganeti VM cluster Manager for High-Availability via Automated Master Node Failover

DIPLOMA THESIS

Nikolaos A. Parasyris

Advisor: Nectarios Koziris
Professor NTUA

. . . . .	. . . . .	. . . . .
Nectarios Koziris	Nikolaos Papaspyrou	Georgios Goumas
Professor NTUA	As. Proffesor NTUA	As. Professor NTUA

Athens, July 2017

. . . . . .

Νικόλαος Α. Παρασύρης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Στη σύγχρονή εποχή, η ζήτηση για υπηρεσίες διαδικτύου και υπολογιστικού νέφους αυξάνεται διαρκώς. Ένα βασικό μοντέλο υπηρεσίας νέφους είναι η Υποδομή ως Υπηρεσία, οπού πάροχοι προσφέρουν υπολογιστικές υποδομές ως πόρους στους χρήστες. Η συνεχής λειτουργία τέτοιων συστημάτων αποτελεί βασικό κριτήριο για τους χρήστες και άρα είναι σημαντικό για λόγους ανταγωνισμού και μεγιστοποίησης κέρδους. Ένα ιδανικό σύστημα θα μπορούσε πάντα να προσφέρει συνεχή λειτουργία. Δυστυχώς, η απόλυτη συνεχής λειτουργία είναι ανέφικτη, καθώς στη πραγματικότητα η πιθανότητα αποτυχίας πολλαπλών συστατικών του συστήματος που οδηγούν σε προσωρινή διακοπή της υπηρεσίας είναι μη μηδενική. Επομένως, η μεγιστοποίηση του χρόνου λειτουργίας, ένα χαρακτηριστικό γνωστό και ως υψηλή διαθεσιμότητα, είναι ιδιαίτερα σημαντική.

Το σύστημα διαχείρισης εικονικών μηχανών Google *Ganeti* χρησιμοποιείται σε συστήματα υποδομών ως υπηρεσία που προσφέρουν εικονικές μηχανές ως πόρους. Ένα σύστημα Ganeti τρέχει σε πολλαπλούς κόμβους, που σχηματίζουν μία συστοιχία. Ένας μοναδικός κόμβος, ο κύριος κόμβος, είναι υπεύθυνος για τη διαχείριση της συστοιχίας και την επεξεργασία των δεδομένων διαμόρφωσης του. Εάν ο κύριος κόμβος γίνει ανενεργός, ένας διαφορετικός κόμβος χρειάζεται να αναλάβει καθήκοντα κύριου κόμβου ώστε να συνεχιστεί η λειτουργία της συστοιχίας. Αυτή η διαδικασία λέγεται μετάπτωση κύριου κόμβου και στο Ganeti εκτελείται από έναν διαχειριστή του συστήματος. Η πολιτική εκτέλεσης μετάπτωσης κύριου κόμβου από διαχειριστή, οδηγεί σε αύξηση του χρόνου μη λειτουργίας και εισάγει το παράγοντα του ανθρώπινου λάθους.

Πρόθεση μας είναι να αυξήσουμε τη διαθεσιμότητα του συστήματος Ganeti υλοποιώντας μια αυτόματη μέθοδο μετάπτωσης κύριου κόμβου. Στόχοι είναι η διασφάλιση της συνοχής των δεδομένων διαμόρφωσης και η ορθή λειτουργία υπό συνθήκες διαμέρισης συστοιχίας. Το etcd, ένα αξιόπιστο κατανεμημένο σύστημα αποθήκευσης κλειδιών-τιμών, χρησιμοποιήθηκε ως μέσο αποθήκευσης των δεδομένων διαμόρφωσης του Ganeti. Επιπλέον, υλοποιήθηκε ένας μηχανισμός που, αυτόματα εντοπίζει αποτυχίες του κύριου κόμβου και εκκινεί τη διαδικασία μετάπτωσης σε κατάλληλο κόμβο. Η υλοποίηση μας ικανοποιεί τα παραπάνω κριτήρια και ολοκληρώνει τη διαδικασία εντός ενός μικρού χρονικού διαστήματος

Λέξεις κλειδιά: συστοιχία, υψηλή διαθεσιμότητα, Ganeti, Etcd, αυτόματη μετάπτωση κύριου κόμβου

Abstract

In modern times the demand for web and cloud computing services is ever growing. Infastructure as a Service (IaaS) is a basic cloud-service model where providers offer computing infrastructure as a service to subscribers. Continuous service of such systems is a basic client criterion and therefore it is important for competitiveness and profit maximizing. An ideal system would always provide continues service. Unfortunately this is not feasible, due to the possibility of multiple node failures, cluster partitions and network system failing. Therefore being as close as possible to continues service, a feature called high-availability, is of high importance.

Google's *Ganeti* virtual machine (VM) cluster manager is such a system that can be used in an IaaS to offer VMs as a resource. Ganeti has a single node as master, that is the only node allowed to run cluster-wide commands and modify the cluster's configuration data. If this node fails, for the ganeti service to continue to operate another node has to take over as master, a procedure called master-failover. In a master node failure scenario, an administrator has to manually execute a master failover operation, which substantially increases the downtime of the service and induces the human error element. Our objective is to extend ganeti for high-availability with an automated master failover procedure that will, in any scenario, ensure configuration data consistency.

After a thorough inspection of the current implementation of Ganeti, we concluded that the checks executed during configuration data distribution and master failover do not ensure configuration data consistency. We decided to migrate the configuration data on *etcd*, a distributed reliable key-value store, that will ensure configuration data consistency. In addition, we implemented a mechanism that will detect a master node failure or a cluster partition and initiate a master failover if needed. Our implementation operates correctly under cluster partitions, detects a master node failure and completes a master failover within a short period of time while ensuring configuration data consistency.

Keywords: cluster, high-availability, VM cluster manager, Ganeti, automated master-failover, etcd.

Preface

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Νεκτάριο Κοζύρη για τα ερεθίσματα, την έμπνευση και την καθοδήγηση που μου προσέφερε. Θέλω ακόμη να ευχαριστήσω τον Βαγγέλη Κούκη, για την άρτια συνεργασία μας, τις γνώσεις, τις ιδέες και την υποστήριξή του κατά την εκπόνηση αυτής της εργασίας. Ένα ιδιαίτερο ευχαριστώ οφείλω και στον Χρήστο Σταυρακάκη, για την υπομονή, τον χρόνο και τις πολύτιμες συμβουλές και υποδείξεις του. Τέλος, ευχαριστώ θερμά την οικογένειά μου για την στήριξή της, τόσο κατά την διάρκεια αυτής της διπλωματικής, όσο και σε όλα τα ακαδημαϊκά μου χρόνια.

Νικόλαος Παρασύρης
Ιούλιος 2017

Contents

Περίληψη **vii**

Abstract **ix**

Preface **xi**

List of figures **xv**

Επέκταση του Συστήματος Διαχείρισης Εικονικών Μηχανών Google Ganeti για την Παροχή Υψηλής Διαθεσιμότητας με Αυτόματη Μετάπτωση του Κύριου Κόμβου **1**

1 Εισαγωγή 1

2 Υπόβαθρο 3

2.1 Εισαγωγή στη θεωρία συστοιχιών και προκλήσεις 3

2.2 Το σύστημα Ganeti 8

2.3 Ο αλγόριθμος Raft 10

2.4 Το σύστημα Etcd 12

3 Ανάλυση του συστήματος Ganeti και σχεδιασμός 12

3.1 Αποθήκευση και διαμοιρασμός δεδομένων διαμόρφωσης . . . 13

3.2 Μετάπτωση κύριου κόμβου 14

3.3 Προτεινόμενες αλλαγές 15

4 Υλοποίηση 16

4.1 Διαχείριση της etcd συστοιχίας 16

4.2 Μεταφορά των δεδομένων διαμόρφωσης στο etcd 17

4.3 Μηχανισμός αυτόματης μετάπτωσης κύριου κόμβου 17
5 Αξιολόγηση και μελλοντικές δυνατότητες 19
5.1 Αξιολόγηση 19
5.2 Μελλοντικές δυνατότητες 19

1 Introduction **21**
1.1 Problem 21
1.2 Incentives 21
1.3 Shortcomings 22
1.4 Objective 22
1.5 Design and evaluation 22

2 Background **25**
2.1 Introduction to Cluster Theory and Challenges 25
2.1.1 Split-brain 29
2.1.2 CAP theorem 33
2.2 Ganeti 37
2.2.1 Ganeti cluster architecture and node roles 40
2.2.2 Ganeti daemons 41
2.2.3 Ganeti CLI 45
2.3 Raft and Etcd 48
2.3.1 Raft 48
2.3.2 Etcd 52

3 Design **59**
3.1 Ganeti configuration store and distribution policy 59
3.2 Ganeti master failover 62
3.3 Proposed changes 65

4 Implementation **67**
4.1 Managing the etcd cluster 67
4.2 Migrating ganeti configuration to etcd 74
4.3 Automatic mechanism for master failover 77

5 Evaluation & Discussion **83**

6 Future Work & Conclusions **89**

Bibliography **93**

List of figures

2.1 Example of a high availability cluster 27
2.2 Example of a load-balancing cluster 28
2.3 Example of a high-performance cluster 29
2.4 A two-node cluster setup 30
2.5 CAP theorem 34
2.6 Example of a ganeti cluster 38
2.7 Interactions between the main ganeti daemons 45
2.8 Replicated state machine architecture 49
2.9 Raft server roles and transitions between them 50

1 Εισαγωγή

Το πρόβλημα υπό εξέταση είναι η επέκταση του συστήματος διαχείρισης εικονικών μηχανών Google Ganeti[RK14] για την παροχή υψηλής διαθεσιμότητας με αυτόματη μετάπτωση του κύριου κόμβου. Το Ganeti είναι ένα σύστημα Υποδομή ως Υπηρεσία (Infrastructure as a Sevice) που προσφέρει εικονικές μηχανές ως πόρους στους χρήστες. Εικονικές μηχανές είναι διάσπαρτες σε πολλούς κόμβους, που σχηματίζουν μία υπολογιστική συστοιχία (computing cluster). Ένας συγκεκριμένος κόμβος, ο κύριος κόμβος, είναι υπεύθυνος για την διαχείριση της συστοιχίας. Σε περίπτωση σφάλματος στον κύριο κόμβο τότε, προκειμένου να συνεχιστεί η κανονική λειτουργία του συστήματος, ένας άλλος κόμβος πρέπει να αναλάβει καθήκοντα κύριου κόμβου. Η λειτουργία αυτή ονομάζεται μετάπτωση κύριου κόμβου. Στην υπάρχουσα υλοποίηση του Ganeti η μετάπτωση κύριου κόμβου γίνεται χειροκίνητα, όπου ένας διαχειριστής του συστήματος αναλαμβάνει την ευθύνη να εντοπίσει το σφάλμα κύριου κόμβου και να προχωρήσει μετατρέποντας έναν άλλον κόμβο σε κύριο. Η χειροκίνητη προσέγγιση εισάγει διάφορα προβλήματα και διάφορες προσεγγίσεις για αυτοματοποίηση αυτής της διαδικασίας κρίθηκαν ανεπαρκείς και προβληματικές.

Στη σύγχρονη εποχή, όπου οι υπηρεσίες διαδικτύου και υπολογιστικού νέφους διαρκώς επεκτείνονται, η ζήτηση για συστήματα Υποδομής ως Υπηρεσία αυξάνεται συνεχώς. Χρήστες επιθυμούν να βασίσουν τις εφαρμογές τους σε υπάρχοντα συστήματα που αναλαμβάνουν λεπτομέρειες υποδομών όπως αντίγραφα ασφάλειας, κατανομή δεδομένων, επεκτασιμότητα, κ.α. Η συνεχής λειτουργία τέτοιων συστημάτων κρίνεται άκρως σημαντική από τους χρήστες. Οι πάροχοι προσπαθούν να βελτιστοποιήσουν τα συστήματα τους για λόγους ανταγωνισμού και μεγιστοποίησης κέρδους. Επομένως, οι προγραμματιστές υλοποιούν διάφορες τεχνικές[vV14] ώστε να μεγιστοποιήσουν την διαθεσιμότητα των συστημάτων τους.

Όπως αναφέρθηκε, η μετάπτωση κύριου κόμβου στο Ganeti γίνεται χειροκίνητα. Ένας διαχειριστής πρέπει να εντοπίσει το σφάλμα στον κύριο κόμβο και να βρει έναν κατάλληλο κόμβο όπου θα εκκινήσει την διαδικασία μετάπτωσης κύριου κόμβου. Η διαδικασία αυτή διαρκεί ένα σεβαστό χρονικό διάστημα, όπου η υπηρεσία δεν είναι διαθέσιμη, καθώς και εισάγει τον παράγοντα του ανθρώπινου λάθους. Διάφορες προσεγγίσεις για την επίλυση του ζητήματος κρίθηκαν ανεπαρκείς και προβληματικές, καθώς είτε οδηγούσαν σε απώλεια δεδομένων διαμόρφωσης είτε δεν διαχειρίζονταν

σωστά σενάρια διαμέρισης της συστοιχίας και οδηγούσαν σε καταστάσεις split-brain. Η κατάσταση split-brain είναι αποτέλεσμα διαμέρισης συστοιχίας, όπου διακομιστές δεν μπορούν να επικοινωνήσουν μεταξύ τους και να συγχρονίσουν δεδομένα και λειτουργίες. Παρ' όλα αυτά συνεχίζουν και λειτουργούν αυθαίρετα δημιουργώντας δύο διαφορετικά σύνολα δεδομένων διαμόρφωσης και αναλαμβάνουν κοινούς πόρους. Η συμπεριφορά αυτή οδηγεί σε καταστροφικά αποτελέσματα για τη υγεία της συστοιχίας.

Σκοπός μας είναι η επίτευξη υψηλής διαθεσιμότητας στο σύστημα Ganeti με την ενσωμάτωση ενός μηχανισμού αυτόματης μετάπτωσης κύριου κόμβου. Η υλοποίηση μας πρέπει να διασφαλίζει τη συνοχή των δεδομένων διαμόρφωσης και να λειτουργεί ορθά σε σενάρια διαμέρισης συστοιχίας.

Προτείνουμε την ενσωμάτωση του etcd[Devc] στο Ganeti, ως μέσο αποθήκευσης των δεδομένων διαμόρφωσης. Το etcd είναι ένα αξιόπιστο κατανεμημένο σύστημα αποθήκευσης κλειδιών-τιμών, που συνήθως χρησιμοποιείται για την αποθήκευση κρίσιμων δεδομένων ενός κατανεμημένου συστήματος. Το etcd θα προσφέρει μία ενιαία και συνεπής εικόνα των δεδομένων διαμόρφωσης σε όλους τους κόμβους της συστοιχίας. Επιπλέον, η πολιτική απαρτίας (quorum)[1] του etcd θα επιτρέψει το σωστή διαχείριση διαμερίσεων της συστοιχίας αποφεύγοντας καταστάσεις split-brain. Επιπροσθέτως, θα υλοποιηθεί ένας μηχανισμός που αυτόματα θα ανιχνεύει βλάβες κύριου κόμβου και θα εκκινεί την διαδικασία μετάπτωσης κύριου κόμβου σε κατάλληλο υποψήφιο κόμβο. Υποψήφιοι κόμβοι στο Ganeti είναι επιλεχθείς κόμβοι οι οποίοι μελλοντικά ίσως αναλάβουν τον ρόλο του κύριου κόμβου. Η υλοποίηση μας έχει έναν βασικό περιορισμό, που είναι απόρροια της πολιτικής απαρτίας του etcd. Ο περιορισμός αυτός είναι ότι, προκειμένου η υλοποίηση μας να λειτουργεί κανονικά ανά πάσα στιγμή πρέπει η πλειονότητα των υποψήφιων κόμβων της συστοιχίας να είναι διαθέσιμοι και η επικοινωνία μεταξύ τους υγιής. Ο περιορισμός αυτός είναι αναγκαίος ώστε να διαχειριζόμαστε σωστά τις διαμερίσεις συστοιχίας. Η υλοποίηση μας ελέγχθηκε σε συστοιχία τριών κόμβων όπου η αυτόματη μετάπτωση κύριου κόμβου ολοκληρώθηκε εντός 40 δευτερολέπτων ενώ παράλληλα η συνοχή των δεδομένων διαμόρφωσης διασφαλίζεται.

[1] Η πολιτική απαρτίας δηλώνει τον αριθμό ψήφων που μια κατανεμημένη συναλλαγή πρέπει να αποκτήσει ώστε να εκτελεστεί. Πολιτική απαρτίας χρησιμοποιείται για να διασφαλίσει την συνεκτική λειτουργία ενός κατανεμημένου συστήματος.

2 Υπόβαθρο

Το κεφάλαιο αυτό περιέχει βασική θεωρία για υπολογιστικές συστοιχίες και θα αναλυθούν προκλήσεις που προκύπτουν σε υπολογιστικά συστήματα που χρησιμοποιούνται σε συστοιχίες. Επιπλέον θα αναλυθούν δύο εργαλεία συστοιχίας που θα χρησιμοποιηθούν στην υλοποίηση μας, το Ganeti και το etcd.

2.1 Εισαγωγή στη θεωρία συστοιχιών και προκλήσεις

Η υπολογιστική συστοιχία (computing cluster) είναι ένας τύπος παράλληλου/κατανεμημένου υπολογιστικού συστήματος, που αποτελείται από μία συλλογή υπολογιστικών μονάδων, γνωστοί και ως κόμβοι, που συνεργάζονται μεταξύ τους ώστε να προσφέρουν έναν ενιαίο υπολογιστικό πόρο[Bak01]. Οι λειτουργίες των κόμβων οργανώνονται από ένα επίπεδο λογισμικού, το οποίο είναι ενεργό σε κάθε κόμβο και επιτρέπει στους χρήστες να χειρίζονται τη συστοιχία ως μία ενιαία συνεκτική υπολογιστική μονάδα. Οι κόμβοι, συνήθως, συνδέονται και επικοινωνούν μεταξύ τους μέσω ενός γρήγορου τοπικού δικτύου.

Οι υπολογιστικές συστοιχίες δεν εφευρέθηκαν από κάποιο πάροχο, αλλά δημιουργήθηκαν από την ανάγκη χρηστών για περισσότερους υπολογιστικούς πόρους αλλά και αντίγραφα ασφαλείας των δεδομένων τους. Από την δημιουργία τους, στις αρχές του 1960, μέχρι σήμερα η χρήση και η ανάπτυξη των υπολογιστικών συστοιχιών αυξάνει διαρκώς λόγω των πλεονεκτημάτων που προσφέρουν. Τα πλεονεκτήματα αυτά είναι[enga][engb]:

- απόδοση κόστους: Η σχέση υπολογιστικής δύναμης και ταχύτητας των συστοιχιών με το κόστος κατασκευής τους είναι αρκετά αποδοτική σε σύγκριση με άλλες τεχνικές, όπως η δημιουργία μεγάλων κεντρικών υπολογιστών.
- Ταχύτητα επεξεργασίας: Πολλαπλές γρήγορες υπολογιστικές μονάδες συνεργάζονται και προσφέρουν ενιαία υψηλή ταχύτητα επεξεργασίας.
- Βελτιωμένη υποδομή δικτύου: Οι κόμβοι συνδέονται με διάφορες τοπολογίες δικτύου ώστε ελαχιστοποιηθεί το κόστος επικοινωνίας και να αποφευχθούν σημεία συμφόρησης.

- Ευελιξία: Οι κόμβοι της συστοιχίας μπορούν να τροποποιηθούν σχετικά εύκολα ώστε να προσφέρουν νέες υπηρεσίες ή να χρησιμοποιήσουν νέες τεχνολογίες.
- Υψηλή διαθεσιμότητα πόρων: Εάν κάποιο συστατικό της συστοιχίας παρουσιάσει πρόβλημα, τότε τα υπόλοιπα συστατικά συνεχίζουν να λειτουργούν, καλύπτουν το κενό και προσφέρουν μια συνεχής λειτουργία.

Μία συστοιχία σχεδιάζεται με διαφορετικές προτεραιότητες και χαρακτηριστικά ανάλογα με τι υπηρεσία θα προσφέρει στους χρήστες. Οι υπολογιστικές συστοιχίες μπορούν να κατηγοριοποιηθούν σε τέσσερις ομάδες ανάλογα με τις υπηρεσίες που προσφέρουν. Οι ομάδες αυτές είναι[Har99]:

- Συστοιχίες αποθήκευσης: Προσφέρουν ένα ενιαίο και συνεπής σύστημα αρχείων σε όλους τους κόμβους και τους επιτρέπουν να εκτελούν ταυτόχρονα λειτουργίες ανάγνωσης και εγγραφής στο κοινό σύστημα αρχείων. Λειτουργίες όπως αντίγραφα ασφαλείας και αποκατάσταση καταστροφών απλοποιούνται ενώ η ανάγκη για περιττά αντίγραφα εξαλείφεται Επιπλέον, η διαχείριση τέτοιων συστοιχιών είναι ευκολότερη καθώς η εγκατάσταση και διόρθωση εφαρμογών περιορίζεται σε ένα σύστημα αρχείων.
- Συστοιχίες υψηλής διαθεσιμότητας: Ελαχιστοποιούν τον χρόνο διακοπής των εφαρμογών που υποστηρίζονται από τη συστοιχία, εξαλείφοντας μεμονωμένα σημεία αποτυχίας. Αυτό επιτυγχάνεται με λογισμικό υψηλής διαθεσιμότητας που χρησιμοποιεί πλεονάζοντες υπολογιστικούς κόμβους. Σφάλματα λογισμικού/υλικού σε έναν κόμβο εντοπίζονται από το λογισμικό υψηλής διαθεσιμότητας που στη συνέχεια εκκινεί τις εφαρμογές που επηρεάστηκαν σε άλλους κόμβους. Η λειτουργία αυτή είναι γνωστή και ως μετάπτωση. Η διατήρηση της ακεραιότητας και συνέπειας των δεδομένων μεταξύ κόμβων είναι αναγκαία ώστε να μην υποδαυλιστεί η λειτουργία μίας εφαρμογής έπειτα από μετάπτωση. Υπάρχουν διάφορες αρχιτεκτονικές συστοιχιών υψηλής διαθεσιμότητας ανάλογα με τον αριθμό των κόμβων που είναι ενεργοί, δηλαδή φιλοξενούν εφαρμογές, και τον αριθμό των κόμβων που είναι παθητικοί, δηλαδή περιμένουν να αναλάβουν εφαρμογές έπειτα από μία μετάπτωση[Ste01].
- Συστοιχίες εξισορρόπησης φορτίου: Το συνολικό φορτίο διαμοιράζεται μεταξύ των κόμβων της συστοιχίας ώστε να αυξηθεί η ταχύτητα διεκπεραίωσης. Ο αλ-

γόριθμος που διαμοιράζει το φορτίο μπορεί να είναι απλός, για παράδειγμα απλή μέθοδος κυκλικής ανάθεσης σε περίπτωση ενός διακομιστή διαδικτύου, ή αρκετά πιο περίπλοκος, για παράδειγμα όταν το φορτίο έχει πολλαπλούς επιστημονικούς υπολογισμούς. Εάν κάποιος κόμβος της συστοιχίας είναι μη λειτουργικός τότε δεν στέλνονται πλέον αιτήματα σε αυτόν μέχρι να λυθεί το ζήτημα.

- Συστοιχίες υψηλής απόδοσης: Εφαρμογές εκτελούνται παράλληλα σε πολλαπλούς κόμβους, όπου ο καθένας αναλαμβάνει ένα ανεξάρτητο υπολογιστικό κομμάτι της εφαρμογής και το διεκπεραιώνει. Ένας κεντρικός κόμβος είναι υπεύθυνος στο να διασπά το αρχικό φορτίο σε ανεξάρτητα φορτία τα οποία κατανέμει στους κόμβους και στη συνέχειά συλλέγει και συγχωνεύει τα αποτελέσματα.

Split-brain

Μία από τις βασικές προκλήσεις που αντιμετωπίζουν συστήματα που αναπτύσσονται σε υπολογιστικές συστοιχίες είναι να διαχειρίζονται σωστά τις διαμερίσεις συστοιχίας ώστε να μην οδηγηθούν σε κατάσταση split-brain. Η διαμέριση συστοιχίας προκύπτει από σφάλματα στην επικοινωνία μεταξύ των κόμβων με αποτέλεσμα την διαίρεση της συστοιχίας σε υποσυστοιχίες. Οι κόμβοι εντός μίας υποσυστοιχίας μπορούν και επικοινωνούν μεταξύ τους αλλά αδυνατούν να επικοινωνήσουν με οποιονδήποτε κόμβο εκτός της υποσυστοιχίας. Η κατάσταση split-brain είναι όταν διαφορετικές υποσυστοιχίες συνεχίζουν να λειτουργούν χωρίς να λάβουν υπόψη τους την παρουσία των άλλων υποσυστοιχιών. Έτσι δημιουργούνται διαφορετικά σύνολα δεδομένων, που αργότερα δεν γίνεται να συγχωνευθούν σε ένα ενιαίο.

Το split-brain είναι απόρροια της απόφασης κάθε υποσυστοιχίας να συνεχίσει να λειτουργεί θεωρώντας τις υπόλοιπες υποσυστοιχίες ανενεργές. Όμως, η ανικανότητα επικοινωνίας με έναν κόμβο δεν οδηγεί σε ασφαλή συμπεράσματα για την κατάσταση αυτού του κόμβου. Ο κόμβος μπορεί να είναι ανενεργός λόγω προβλήματος ή η επικοινωνία μεταξύ τους να είναι προβληματική. Επομένως, η απόφαση μίας υποσυστοιχίας να θεωρήσει ανενεργές τις υποσυστοιχίες με τις οποίες δεν μπορεί να επικοινωνήσει, είναι αυθαίρετη. Εάν πολλαπλές υποσυστοιχίες λάβουν την ίδια απόφαση τότε θα λειτουργούν ταυτόχρονα χωρίς όμως να μπορούν να συγχρονίσουν τα δεδομένα τους, γεγονός που οδηγεί σε κατάσταση split-brain.

Ένας απλός τρόπος αντιμετώπισης των split-brain είναι να μειωθούν οι περιπτώσεις διαμέρισης της συστοιχίας. Αυτό μπορεί να πραγματοποιηθεί υλοποιώντας πολλαπλά ανεξάρτητα δίκτυα επικοινωνίας μεταξύ των κόμβων. Έτσι, εξαλείφονται μεμονωμένα σημεία αποτυχίας του δικτύου. Παρ' όλα αυτά, η διαμέριση της συστοιχίας είναι πιθανή, άρα τεχνικές διαχείρισης διαμερίσεων έχουν αναπτυχθεί ώστε να αποφευχθούν καταστάσεις split-brain.

Οι τεχνικές διαχείρισής διαμερίσεων για την αποφυγή split-brain μπορούν να κατηγοριοποιηθούν σε δύο ομάδες[Ske85]:

- αισιόδοξες: Οι προσεγγίσεις αυτές επιτρέπουν στις υποσυστοιχίες να συνεχίσουν να λειτουργούν κανονικά, αποθηκεύοντας όμως επιπλέον πληροφορίες για κάθε λειτουργία που εκτελούν. Όταν αποκατασταθούν οι διαμερίσεις, τότε οι επιπλέον αποθηκευμένες πληροφορίες χρησιμοποιούνται ώστε τα διαφορετικά σύνολα δεδομένων να ενωθούν σε ένα ενιαίο και συνεπές σύνολο δεδομένων.

- απαισιόδοξες: Οι προσεγγίσεις αυτές περιορίζουν την λειτουργία και την πρόσβαση σε διαμοιραζόμενους πόρους των υποσυστοιχιών, με σκοπό να διασφαλίσουν την συνέπεια των δεδομένων της συστοιχίας. Στις προσεγγίσεις αυτές, οι δύο βασικές τεχνικές που χρησιμοποιούνται είναι η τεχνική απαρτίας και το fencing.

Το fencing αποτρέπει έναν κόμβο να χρησιμοποιήσει διαμοιραζόμενους πόρους. Μπορεί να γίνει σε δύο διαφορετικά επίπεδα. Σε επίπεδο κόμβου η τεχνική fencing εμποδίζει τον κόμβο να χρησιμοποιήσει οποιοδήποτε διαμοιραζόμενο πόρο χωρίς εξαίρεση. Ένας συνήθης τρόπος που επιτυγχάνεται αυτό είναι τερματίζοντας τελείως τη λειτουργία του κόμβου, μία τεχνική γνωστή και ως STONITH (Shoot The Other Node In The Head). Σε επίπεδο πόρου, η τεχνική εντοπίζει ποιους διαμοιραζόμενους πόρους μπορεί ο κόμβος να χρησιμοποιήσει και αποτρέπει την πρόσβαση του μόνο σε αυτές. Και τα δύο επίπεδα απαιτούν επιπλέον υλικό και λογισμικό για να λειτουργήσουν. Συνήθως το fencing εκκινείται από έναν κόμβο με στόχο έναν άλλον κόμβο, με τον οποίο δεν μπορεί να επικοινωνήσει και άρα θέλει να βεβαιωθεί ότι δεν θα χρησιμοποιήσει διαμοιραζόμενους πόρους. Υπάρχει όμως περίπτωση το fencing να εκκινείται από έναν κόμβο για να εμποδίσει τον εαυτό του να χρησιμοποιήσει διαμοιραζόμενους

πόρους. Η τεχνική αυτή ονομάζεται self-fencing, αλλά είναι σφαλερή καθώς βασίζεται σε έναν πιθανώς προβληματικό κόμβο να αποφασίσει και να εκτελέσει την λειτουργία του self-fencing σωστά.

Η τεχνική του fencing δεν επαρκεί, καθώς χωρίς επιπλέον εργαλεία προβληματικές καταστάσεις μπορεί να προκύψουν. Συγκεκριμένα, είναι πιθανών δύο κόμβοι Α και Β, που δεν μπορούν να επικοινωνήσουν μεταξύ τους, να αποφασίσουν να εφαρμόσουν fencing μεταξύ τους. Δηλαδή ο κόμβος Α στον Β και ο Β στον Α. Αν αυτό γίνει, τότε και οι δύο κόμβοι θα αποκλειστούν από τους διαμοιραζόμενους πόρους, και μάλιστα αν γίνει fencing σε επίπεδο κόμβου με επανεκκίνηση κόμβου τότε το σύστημα θα εισέλθει σε έναν αέναο κύκλο επανεκκινήσεων μεταξύ των δύο κόμβων. Για να λυθούν τέτοια σενάρια χρειάζεται μία επιπλέον τεχνική, η τεχνική απαρτίας.

Η τεχνική απαρτίας[Cou01] ουσιαστικά είναι μία μέθοδος που επιλύει το αμοιβαίο δίλημμα fencing. Το βασικό πρόβλημα είναι να επιλεχθεί μία και μόνο μία υποσυστοιχία που θα συνεχίσει να λειτουργεί επιβάλλοντας την τεχνική fencing στους υπόλοιπους. Η απόφαση αυτή πρέπει να είναι ίδια σε όλες τις υποσυστοιχίες χωρίς να υπάρχει επικοινωνία μεταξύ τους. Η συνήθης λύση σε αυτό το πρόβλημα είναι η πλειοψηφία. Κάθε υποσυστοιχία αριθμεί τα μέλη του και εάν αποτελούν πλειοψηφία του συνολικού αριθμού μελών της συστοιχίας τότε εφαρμόζει fencing στους υπόλοιπους και συνεχίζει να λειτουργεί. Απαιτώντας πλειοψηφία η τεχνική βεβαιώνει ότι το πολύ μία υποσυστοιχία θα συνεχίσει να λειτουργεί. Βέβαια είναι πιθανών καμία υποσυστοιχία να μην ικανοποιεί την συνθήκη της πλειοψηφίας και άρα όλες οι υποσυστοιχίες να ”παγώσουν” και να μην υπάρξει πρόοδος στη συστοιχία μέχρι η διαμέριση να επιλυθεί.

Η χρήση τεχνικών fencing και απαρτίας συνιστάται ώστε τα σενάρια διαμέρισης συστοιχίας να αντιμετωπίζονται σωστά και να αποφεύγονται καταστάσεις split-brain.

Θεώρημα CAP

Το θεώρημα CAP[Bro][Tha][Gil02] περιορίζει το σχεδιαστικό χώρο των προγραμματιστών συστημάτων που αναπτύσσονται σε συστοιχίες. Συγκεκριμένα, ορίζει ότι ένα σύστημα δεν μπορεί ταυτόχρονα να ικανοποιεί σε απόλυτο βαθμό και τις τρεις παρακάτω ιδιότητες.

- Συνοχή δεδομένων (**C**onsistency)

- Διαθεσιμότητα (**A**vailability)
- Ανοχή διαμερίσεων (**P**artition tolerance)

Απόλυτη συνοχή των δεδομένων και διαθεσιμότητα της υπηρεσίας είναι δυνατή όταν η συστοιχία δεν βρίσκεται σε κατάσταση διαμέρισης. Εάν παρουσιαστεί διαμέριση τότε ο σχεδιαστής πρέπει να θυσιάσει είτε τη συνοχή των δεδομένων του επιτρέποντας στο σύστημα να συνεχίσει τη λειτουργία του, είτε να περιορίσει τη διαθεσιμότητα του συστήματος του ώστε να εξασφαλίσει τη συνοχή των δεδομένων. Υλοποιήσεις που εξασφαλίζουν έναν βαθμό συνοχής δεδομένων και διαθεσιμότητας του συστήματος υπό συνθήκες διαμέρισης είναι δυνατές. Επειδή σε πραγματικά συστήματα οι διαμερίσεις είναι αναπόφευκτες, ο σχεδιαστής οφείλει να γνωρίζει πως λειτουργεί το σύστημα του σε τέτοιες καταστάσεις, και πιθανώς να τροποποιήσει τη συμπεριφορά του ώστε να εξασφαλίζει τον βαθμό συνοχής ή διαθεσιμότητας που απαιτεί η υπηρεσία που προσφέρει.

2.2 Το σύστημα Ganeti

Το Ganeti είναι ένα σύστημα διαχείρισης εικονικών μηχανών σε συστοιχία. Το Ganeti διευκολύνει τη διαχείριση της συστοιχίας και των εικονικών μηχανών επιτελώντας λειτουργίες όπως:

- δημιουργία δίσκων
- εγκατάσταση λειτουργικών συστημάτων για τις εικονικές μηχανές
- έναρξη/απενεργοποίηση των εικονικών μηχανών και μετάπτωση μεταξύ κόμβων

Υποστηρίζει διάφορες τεχνικές και τεχνολογίες για τη δημιουργία εικονικών μηχανών, δίσκων, δικτύου μεταξύ των μηχανών, κ.α. Μερικά από τα πλεονεκτήματα του συστήματος Ganeti είναι η απλή και ευνόητη αρχιτεκτονική, η κλιμακοσιμοτητα του, ο υψηλός βαθμός ρύθμισης που επιτρέπει και ότι διαχειρίζεται εύκολα από μικρό αριθμό διαχειριστών.

Η λειτουργικότητα που υποστηρίζει και οι ευθύνες που έχει ένας κόμβος συστοιχίας Ganeti εξαρτώνται από τον ρόλο του. Οι πιο συχνοί, και σημαντικοί για την υλοποίηση μας, ρόλοι είναι:

- Κύριος κόμβος: υπεύθυνος για τη διαχείριση της συστοιχίας και να εκτελεί εντολές από τους διαχειριστές και τους χρήστες. Επιπλέον είναι υπεύθυνος για την τροποποίηση και το διαμοιρασμό των δεδομένων διαμόρφωσης.
- Υποψήφιος κόμβος: είναι υποψήφιος να γίνει κύριος κόμβος εάν παρουσιαστεί σφάλμα στην λειτουργία του τωρινού κύριου κόμβου
- Κανονικός κόμβος: Υπεύθυνος για τη φιλοξενία εικονικών μηχανών και τη εκτέλεση αιτημάτων από τον κύριο κόμβο

Οι κόμβοι μπορεί να αλλάξουν ρόλο με τις κατάλληλες εντολές από τη διεπαφή γραμμής εντολών που έχει υλοποιήσει το Ganeti. Ο ρόλος ενός κόμβοι καθορίζει τη λειτουργικότητα που προσφέρει. Η συνολική λειτουργικότητα του Ganeti έχει διαχωριστεί σε ανεξάρτητες οντότητες, κάθε μία εκ των οποίων έχει ανατεθεί σε έναν δαίμονα. Οι δαίμονες που υλοποιούν τη βασική λειτουργικότητα του Ganeti είναι:

- *ganeti-noded*[Devm]: είναι ενεργός σε όλους τους κόμβους και είναι υπεύθυνος στο να δέχεται μέρος των δεδομένων διαμόρφωσης της συστοιχίας από τον κύριο κόμβο και να τα αποθηκεύει Επιπλέον, εκτελεί εντολές από τον κύριο κόμβο που αφορούν εικονικές μηχανές
- *ganeti-confd*[Devi]: είναι ενεργός μόνο σε υποψήφιους κόμβους και δέχεται μέρος των δεδομένων διαμόρφωσης της συστοιχίας και τα αποθηκεύει. Αυτά τα δεδομένα είναι διαθέσιμα μόνο στους υποψήφιους και στον κύριο κόμβο και είναι αναγκαία για να αναλάβει αργότερα λειτουργία ως κύριος κόμβος.
- *ganeti-wconfd*[Devq]: είναι ενεργός μόνο στον κύριο κόμβο και είναι υπεύθυνος στο να εκτελεί εντολές που τροποποιούν τα δεδομένα διαμόρφωσης της συστοιχίας, να αποθηκεύει τις αλλαγές και να διανέμει τα νέα δεδομένα διαμόρφωσης στους υπόλοιπους κόμβους.
- *ganeti-luxid*[Devk]: είναι ενεργός μόνο στον κύριο κόμβο. Δέχεται όλα τα αιτήματα διαχειριστών και τα εισάγει στην ουρά εργασίας του Ganeti. Επιπλέον

είναι υπεύθυνος για την διαχείριση της ουράς εργασίας και την έναρξη κάθε εργασίας.

- *ganeti-rapid*[Devn]: είναι ενεργός μόνο στον κύριο κόμβο και επιτρέπει σε εξωτερικά εργαλεία να στείλουν αιτήματα στο κύριο κόμβο. Υλοποιεί και παρέχει μια διαπροσωπία και προωθεί και αιτήματα που λαμβάνει στον ganeti-luxid.

Όπως αναφέρθηκε ήδη το Ganeti παρέχει μια διεπαφή γραμμής εντολών την οποία μπορούν να χρησιμοποιήσουν διαχειριστές. Οι εντολές αυτές υλοποιούν όλες τις λειτουργίες που υποστηρίζει το σύστημά Ganeti. Μία εντολή μεταφράζεται σε ένα ή πολλαπλά αιτήματα που προωθούνται στον ganeti-luxid.

2.3 Ο αλγόριθμος Raft

Ο Raft είναι ένας αλγόριθμος συναίνεσης (quorum algorithm)[2] που προσφέρει έναν γενικό τρόπο διανομής μίας μηχανής καταστάσεων σε μία συστοιχία υπολογιστικών συστημάτων, διασφαλίζοντας ότι κάθε κόμβος συμφωνεί στην ίδια σειρά μετάβασης καταστάσεων. Ο Raft σχεδιάστηκε ώστε να είναι ασφαλής υπό όλες τις συνθήκες και διαθέσιμο υπό τυπικές συνθήκες κανονικής λειτουργίας. Επιπλέον προσφέρει ένα πρακτικό υπόβαθρο για την ανάπτυξη συστημάτων, απλοποιώντας την εργασία των προγραμματιστών και είναι εύκολα κατανοητός, ένας από τους βασικούς σχεδιαστικούς στόχους του[Ong13].

Οι αλγόριθμοι συναίνεσης συχνά χρησιμοποιούνται στο πλαίσιο των replicated state machines, το καθένα εκ των οποίων αποτελείται από μία ντετερμινιστική μηχανή καταστάσεων, ένα ημερολόγιο εντολών και μία μονάδα συναίνεσης. Ο αλγόριθμος συναίνεσης είναι μέρος της μονάδας συναίνεσης που είναι υπεύθυνη να αναπαράγει το ημερολόγιο εντολών σε όλες τις οντότητες με όλες τις εντολές στην ακριβώς ίδια σειρά. Οι μηχανές καταστάσεων διαβάζουν τις εντολές από το ημερολόγιο εντολών και τις εκτελούν Εφόσον οι μηχανές είναι ντετερμινιστικές και οι εντολές στο ημερολόγιο έχουν αναπαραχθεί με την ίδια ακριβώς σειρά, όλες οι ενδιάμεσες και η τελική κατάσταση όλων των μηχανών καταστάσεων είναι ίδιες.

[2] Αλγόριθμοι συναίνεσης λύνουν το πρόβλημα της συναίνεσης, όπου όλες οι οντότητες ενός συστήματος πρέπει να συμφωνήσουν σε μία συγκεκριμένη τιμή

Οι κόμβοι σε μία συστοιχία που χρησιμοποιεί Raft έχουν διαφορετικούς ρόλους. Οι ρόλοι αυτοί είναι:

- Ηγέτης: ηγέτης είναι το πολύ ένας κόμβος ανά πάσα στιγμή και είναι ο μοναδικός υπεύθυνος για να διαχειρίζεται και να διανέμει το ημερολόγιο εντολών.
- Κανονικός: στον ρόλο αυτό ο κόμβος δέχεται αιτήματα από τον ηγέτη για εγγραφές στο ημερολόγιο εντολών και τις εκτελεί.
- Υποψήφιος: Εάν ένας κανονικός κόμβος δεν λάβει μήνυμά από τον ηγέτη μέσα σε ένα ορισμένο χρονικό διάστημα τότε γίνεται υποψήφιος και εκκινεί τη διαδικασία εκλογής ηγέτη.

Έχοντας ένα μόνο κόμβο να λειτουργεί ως ηγέτης και να είναι ο μοναδικός υπεύθυνος για την διαχείριση του ημερολογίου εντολών ο Raft απλοποιεί την ροή δεδομένων στο σύστημα και διασπάει το πρόβλημα της συναίνεσης σε δύο βασικές λειτουργίες:

- εκλογή ηγέτη: ο ηγέτης στέλνει περιοδικά μηνύματα σε όλους τους κόμβους ώστε να διατηρήσει την αρχηγεία. Εάν ένας κόμβος δεν λάβει τέτοιο μήνυμα μέσα σε συγκεκριμένο χρονικό διάστημα τότε γίνεται υποψήφιος και εκκινεί τη διαδικασία εκλογής. Για να εκλεχθεί στέλνει μήνυμα σε κάθε κόμβο όπου περιέχει πληροφορίες για τη τελευταία καταχώρηση στο ημερολόγιο εντολών του. Οι κόμβοι των ψηφίζουν αν και μόνο αν κρίνουν ότι ο υποψήφιος έχει τουλάχιστον εξίσου πρόσφατο ημερολόγιο με τους ίδιους. Ένας κόμβος μπορεί να ψηφίσει μόνο μία φορά σε κάθε διαδικασίας εκλογής. Για να μπορέσει ο υποψήφιος να συνεχίσει ως ηγέτης πρέπει να λάβει θετικούς ψήφους από τη πλειοψηφία της συστοιχίας.
- αντιγραφή ημερολογίου εντολών: Η αντιγραφή του ημερολογίου εντολών γίνεται εξ ολοκλήρου από τον ηγέτη. Ο ηγέτης δέχεται αιτήσεις για καταχωρήσεις στο ημερολόγιο. Όταν γίνει αυτό προωθεί την καταχώρηση σε όλα τα μέλη της συστοιχίας. Όταν λάβει θετικές απαντήσεις από τη πλειοψηφία της συστοιχίας ότι δέχτηκαν την καταχώρηση τότε κατοχυρώνει την καταχώρηση, την εκτελεί στην μηχανή καταστάσεων του και προωθεί τα αποτελέσματα στον χρήστη που έκανε την αίτηση.

Η πολιτική πλειοψηφίας κατά τη διάρκεια της εκλογής ηγέτη και της αντιγραφής του ημερολογίου, σε συνδυασμό με την μοναδικότητα του ηγέτη, εξασφαλίζουν ότι οποιαδήποτε καταχώρηση κατοχυρωθεί δεν θα χαθεί ακόμα και αν αλλάξει ο ηγέτης. Επιπλέον όλες κατοχυρωμένες καταχωρήσεις είναι, ή μελλοντικά όταν εισαχθούν θα ειναι, με ακριβώς την ίδια σειρά σε όλα τα ημερολόγια εντολών της συστοιχίας. Επομένως, ο αλγόριθμος Raft επιλύει το πρόβλημα της συναίνεσης.

2.4 Το σύστημα Etcd

Το etcd είναι ένα αξιόπιστο κατανεμημένο σύστημα αποθήκευσης κλειδιών-τιμών, που συνήθως χρησιμοποιείται για την αποθήκευση κρίσιμων δεδομένων ενός κατανεμημένου συστήματος. Σκοπός του είναι να προσφέρει ένα απλό, ασφαλές, γρήγορο και αξιόπιστο μέσο αποθήκευσης σε ένα κατανεμημένο σύστημα. Το etcd υλοποιεί τον αλγόριθμο Raft προκειμένου να εξασφαλίσει τη συνέπεια των δεδομένων που αποθηκεύει. Επιπλέον έχει σχεδιαστεί ώστε να μην επηρεάζεται η ορθότητα του από σφάλματα μελών, ηγέτη, πλειοψηφίας ή δικτύου.

Το etcd προσφέρει διάφορες δυνατότητές για την δημιουργία μίας συστοιχίας etcd, την εισαγωγή νέων μελών, την σύνδεση μεταξύ αυτών, και κυρίως διάφορες παραμετροποιήσεις στη αποθήκευση των κλειδιών. Επιπλέον, οι διαδικασίες αποθήκευσης δεδομένων, αναζήτησης κλειδιών και σύνδεσης μεταξύ μελών έχουν βελτιστοποιηθεί με σκοπό την ταχύτερη ολοκλήρωση αυτών των λειτουργιών.

3 Ανάλυση του συστήματος Ganeti και σχεδιασμός

Στο κεφάλαιο αυτό θα αναλύσουμε πως η τωρινή υλοποίηση του Ganeti επιτελεί κάποιες βασικές λειτουργίες που συνδέονται άμεσα με τη διαδικασία μετάπτωσης κύριου κόμβου. Επιπλέον θα αναλύσουμε γιατί αυτές οι διαδικασίες είναι προβληματικές και απαιτούνται αλλαγές ώστε να υλοποιήσουμε μία αυτόματη διαδικασία μετάπτωσης κύριου κόμβου.

3.1 Αποθήκευση και διαμοιρασμός δεδομένων διαμόρφωσης

Τα δεδομένα διαμόρφωσης μίας συστοιχίας Ganeti αποτελούνται από διάφορα αρχεία που αποθηκεύονται τοπικά στους κόμβους. Το βασικό αρχείο που εμπεριέχει την ολότητα των δεδομένων διαμόρφωσης είναι το config.data. Από αυτό το αρχείο εξάγονται και ομαδοποιούνται συγκεκριμένες πληροφορίες που αποθηκεύονται σε διαφορετικά αρχεία ssconf. Όπως έχει αναφερθεί ο μοναδικός υπεύθυνος για την αποθήκευση και την διανομή αυτών των αρχείων είναι ο ganeti-wconfd που είναι ενεργός στον κύριο κόμβο.

Ο ganeti-wconfd, όταν μία αίτηση οδηγεί σε τροποποίηση των δεδομένων διαμόρφωσης, αποθηκεύει τις αλλαγές τοπικά και έπειτα τις διανέμει στους υπόλοιπους κόμβους της συστοιχίας. Για να το επιτύχει αυτό, κατά την εκκίνηση του δημιουργεί τρία νήματα/εργάτες το καθένα με συγκεκριμένη ευθύνη. Ο πρώτος εργάτης αποθηκεύει τα τροποποιημένα δεδομένα διαμόρφωσης στο τοπικό αρχείο config.data του κύριου κόμβου. Έπειτα εκκινεί τον δεύτερο και τον τρίτο εργάτη. Ο δεύτερος εργάτης διαμοιράζει το αρχείο config.data σε όλους τους υποψήφιους κόμβους της συστοιχίας. Για να το κάνει αυτό διαβάζει τα δεδομένα διαμόρφωσης και εξάγει τη λίστα με τους υποψήφιους κόμβους, στους οποίους στέλνει ένα πακέτο με το νέο τροποποιημένο config.data. Ο τρίτος εργάτης εξάγει από τα δεδομένα διαμόρφωσης τις πληροφορίες που αποθηκεύονται σε όλα τα ssconf αρχεία, τις ομαδοποιεί και τις αποθηκεύει σε μία κατάλληλη δομή δεδομένων. Τη δομή αυτή την στέλνει έπειτα σε όλους τους κόμβους της συστοιχίας.

Είναι σημαντικό να τονίσουμε ότι ο ganeti-wconfd δεν κάνει κανέναν έλεγχο κατά τη διάρκεια της διανομής των δεδομένων διαμόρφωσης. Αυτό είναι ιδιαίτερα σημαντικό για το αρχείο config.data που στέλνει στους υποψήφιους κόμβους. Εφόσον δεν κάνει κανέναν έλεγχο για να διασφαλίσει ότι τα δεδομένα αυτά παραλείφθηκαν από τους υποψήφιους κόμβους ή μία πλειονότητα αυτών είναι δυνατόν να υπάρξει απώλεια δεδομένων διαμόρφωσης όπως θα αναλυθεί στη συνέχεια. Η έλλειψη ελέγχων επιβεβαιώθηκε σε μία δοκιμαστική συστοιχία τριών κόμβων όπου ο κύριος κόμβος συνεχίζει να λειτουργεί κανονικά και να τροποποιεί τα δεδομένα διαμόρφωσης ακόμα και όταν οι άλλοι δύο κόμβοι είναι εκτός λειτουργίας.

3.2 Μετάπτωση κύριου κόμβου

Το Ganeti προσφέρει τη λειτουργία μετάπτωσης κύριου κόμβου η οποία μπορεί να εκκινηθεί από έναν διαχειριστή μέσω της κατάλληλης εντολής από την διεπαφή γραμμής εντολών ή με χρήση της διεπαφής του ganeti-rapi.

Η λειτουργία μετάπτωσης επιτελεί δύο βασικούς ελέγχους, οι οποίοι πρέπει να είναι θετικοί ώστε η λειτουργία να ολοκληρωθεί. Ο πρώτος έλεγχος βεβαιώνει ότι η διαδικασία μετάπτωσης δεν εκκινήθηκε σε κόμβο που είναι ήδη ο κύριος κόμβος της συστοιχίας. Για να το κάνει αυτό διαβάζει από τα τοπικά δεδομένα μετάπτωσης και βρίσκει ποιος είναι ο κύριος κόμβος. Εάν είναι ο ίδιος τότε η διαδικασία μετάπτωσης ακυρώνεται. Εάν δεν είναι τότε συνεχίζει με τον δεύτερο έλεγχο. Ο δεύτερος έλεγχος έχει μορφή ψηφοφορίας. Ο κόμβος, στον οποίο η διαδικασία μετάπτωσης έχει εκκινηθεί, στέλνει μήνυμα σε όλους τους κόμβους της συστοιχίας, την λίστα των οποίων βρίσκει στα τοπικά δεδομένα διαμόρφωσης, ζητώντας να του απαντήσουν με το όνομα του τωρινού κύριου κόμβου που έχει κάθε κόμβος στα τοπικά δεδομένα διαμόρφωσης. Έπειτα συλλέγει τις απαντήσεις και τις ομαδοποιεί σε 3 κατηγορίες. Τις θετικές απαντήσεις, όπου ο κύριος κόμβος που εμπεριέχει η απάντηση είναι ο ίδιος με αυτόν που έχει ο κόμβος στα τοπικά του δεδομένα. Τις αρνητικές απαντήσεις όταν υπάρχει ασυμφωνία και τέλος, η τρίτη ομάδα αποτελείται από τους κόμβους που δεν απάντησαν είτε διότι είναι εκτός λειτουργίας είτε διότι υπάρχει πρόβλημά στο δίκτυο. Για να συνεχίσει η διαδικασία μετάπτωσης ο κόμβος πρέπει να λάβει πλειοψηφία θετικών απαντήσεων.

Ο δεύτερος έλεγχος αν και είναι ψήφος πλειοψηφίας δεν εξασφαλίζει ότι ο κόμβος υπό μετάπτωση έχει τη πιο πρόσφατη εκδοχή των δεδομένων μετάπτωσης. Συγκεκριμένα ο έλεγχος γίνεται αναφορικά με τον κύριο κόμβο που βλέπει κάθε κόμβος και όχι με την έκδοση των δεδομένων διαμόρφωσης. Αυτό σε συνδυασμό με το γεγονός ότι κατά τη διανομή των δεδομένων διαμόρφωσης δεν γίνεται έλεγχος πλειοψηφίας οδηγεί σε πιθανή απώλεια δεδομένων διαμόρφωσης. Ένας κόμβος μπορεί με επιτυχία να ολοκληρώσει τη διαδικασία μετάπτωσης κύριου κόμβου χωρίς να διαθέτει την πιο πρόσφατη έκδοση των δεδομένων διαμόρφωσης

Για παράδειγμά, έστω μία συστοιχία τριών κόμβων, Α, Β, Γ με τον κόμβο Α ως κύριο. Ο κόμβος Γ είναι έκτος λειτουργίας και ο κόμβος Α συνεχίζει να λειτουργεί και να κάνει αλλαγές στα δεδομένα διαμόρφωσης. Έπειτα ο κόμβος Α τίθεται εκτός λειτουργίας

ενώ ο κόμβος Γ εκκινείται. Στον κόμβο Γ μία διαδικασία μετάπτωσης θα περάσει τους ελέγχους και θα ολοκληρωθεί επιτυχημένα, παρότι ο κόμβος διαθέτει παλιά έκδοση των δεδομένων διαμόρφωσης.

Επιπλέον, κατά την διαδικασία μετάπτωσης κύριου κόμβου, ο κόμβος στον οποίο εκτελείται συνδέεται στον τωρινό κύριο κόμβο και σταματάει τη λειτουργία του ως κύριο κόμβο. Εάν δεν μπορεί να συνδεθεί στον κύριο κόμβο ,είτε λόγω προβλήματος δικτύου είτε επειδή είναι εκτός λειτουργίας, τότε ολοκληρώνει τη διαδικασία μετάπτωσης επιτυχώς. Έτσι σε περίπτωση διαμέρισης της συστοιχίας είναι πιθανόν η διαδικασία μετάπτωσης να ολοκληρωθεί σε μία υοσυστοιχία ενώ υπάρχει και άλλος ενεργός κύριος κόμβος σε άλλη υποσυστοιχία. Άρα θα υπάρχουν ταυτόχρονα 2 κύριοι κόμβοι και η συστοιχία θα οδηγηθεί σε κατάσταση split-brain.

3.3 Προτεινόμενες αλλαγές

Έχουμε δει ως τώρα ότι η τωρινή υλοποίηση του Ganeti επιτρέπει απώλεια δεδομένων διαμόρφωσης και μπορεί να οδηγήσει σε κατάσταση split-brain. Η διαδικασία μετάπτωσης εκτελείται από διαχειριστή, ο οποίος επωμίζεται την ευθύνη να αποφευχθούν τα δύο παραπάνω φαινόμενα. Για να το κάνει αυτό πρέπει να εξασφαλίσει ότι ο κόμβος προς μετάπτωση έχει τη πιο πρόσφατη έκδοση δεδομένων διαμόρφωσης και ότι δεν υπάρχει άλλος ενεργός κύριος κόμβος. Η διαδικασία αυτή είναι χρονοβόρα και μπορεί να γίνουν σφάλματα.

Για να αυτοματοποιήσουμε τη διαδικασία αυτή, πρέπει να εξασφαλίσουμε ότι δεν θα υπάρξει απώλεια δεδομένων διαμόρφωσης και επιπλέον ότι οι διαμερίσεις συστοιχίας αντιμετωπίζονται σωστά. Για να το πετύχουμε αυτό θα χρησιμοποιήσουμε το σύστημα etcd ως μέσο αποθήκευσης των δεδομένων διαμόρφωσης του Ganeti συστοιχίας. Το etcd θα εξασφαλίσει μία ενιαία εικόνα των δεδομένων διαμόρφωσης σε όλο τη συστοιχία και επιπλέον η πολιτική πλειοψηφίας που χρησιμοποιεί θα εξασφαλίσει ότι οι διαμερίσεις συστοιχίας δεν θα οδηγήσουν σε split-brain. Επιπλέον, πρέπει να υλοποιηθεί ένας μηχανισμός που θα εντοπίζει σφάλματα του κύριου κόμβου και θα εκκινεί τη διαδικασία μετάπτωσης.

4 Υλοποίηση

Η υλοποίηση μας εισάγει νέες οντότητες στο σύστημα Ganeti και τροποποιεί υπάρχουσες λειτουργίες. Η υλοποίηση μας μπορεί να διασπαστεί σε τρεις ενότητες που επιτελούν διαφορετικό έργο. Αυτές είναι η διαχείριση της etcd συστοιχίας από το σύστημα Ganeti, η αποθήκευση των δεδομένων διαμόρφωσης στο etcd και ο μηχανισμός αυτόματης μετάπτωσης κύριου κόμβου.

4.1 Διαχείριση της etcd συστοιχίας

Το μέλη της etcd συστοιχίας θα είναι οι υποψήφιοι κόμβοι της συστοιχίας Ganeti. Η επιλογή αυτή έγινε διότι στο etcd θα αποθηκεύονται τα δεδομένα διαμόρφωσης για τα οποία υπεύθυνοι είναι οι υποψήφιοι και άρα οι λειτουργίες του etcd πρέπει να εξαρτώνται από αυτούς.

Οι λειτουργίες του Ganeti που αλλάζουν τα μέλη της συστοιχίας πρέπει να τροποποιηθούν ώστε να τροποποιούν κατάλληλα και τα μέλη της etcd συστοιχίας. Συγκεκριμένα, οι λειτουργίες και οι αλλαγές που χρειάστηκαν είναι:

- αρχικοποίηση συστοιχίας Ganeti: η λειτουργία αυτή τροποποιήθηκε έτσι ώστε να δημιουργεί και ένα νέα συστοιχία etcd. Το μοναδικό μέλος και των δύο συστοιχιών είναι ο κόμβος στον οποίο εκκινήθηκε αυτή η διαδικασία.

- προσθήκη/αφαίρεση μέλους: ο κύριος κόμβος του Ganeti μπορεί να προσθέτει και να αφαιρεί μέλη από τη συστοιχία. Αυτές οι λειτουργίες τροποποιήθηκαν έτσι ώστε να διαχειρίζονται και τη etcd συστοιχία. Συγκεκριμένα, όταν προστίθεται ένα μέλος που είναι και υποψήφιος τότε αυτός ο κόμβος προστίθεται και ως μέλος στη etcd συστοιχία. Όταν αφαιρείται ένα μέλος από τη συστοιχία Ganeti που ήταν υποψήφιος κόμβος, τότε αφαιρείται και από τη συστοιχία etcd.

- αλλαγή ρόλων: ο κύριος κόμβος μπορεί να αλλάξει τον ρόλο ενός μέλους του etcd. Όταν προωθεί ένα μέλος στο ρόλο του υποψήφιου τότε αυτό ο κόμβος πρέπει να γίνει μέλος της etcd συστοιχίας. Ομοίως όταν υποβιβάζει έναν υποψήφιο κόμβο σε άλλο ρόλο, ο κόμβος αυτός πρέπει να αφαιρεθεί από τη συστοιχία etcd.

Με τις παραπάνω αλλαγές, το σύστημα Ganeti διαχειρίζεται τη etcd συστοιχία. Χρειάζεται όμως μία επιπλέον οντότητα ώστε σε κάθε κόμβο να εκκινείται η υπηρεσία etcd όταν εκκινείται και η υπηρεσία Ganeti. Η οντότητα αυτή είναι ένας νέος δαίμονας του Ganeti συστήματος, ο ganeti-etcd, ο οποίος εκκινείται πρώτος κατά την εκκίνηση της υπηρεσίας Ganeti. Συγκεκριμένα, ο δαίμονας ganeti-etcd αρχικά ελέγχει αν ο κόμβος είναι μέλος του etcd, δηλαδή υποψήφιος κόμβος στο Ganeti, ή όχι. Εάν είναι τότε ξεκινάει την υπηρεσία etcd ως μέλος της συστοιχίας. Εάν δεν είναι τότε δημιουργεί μία σύνδεση στο etcd συστοιχία ώστε αιτήματα για ανάγνωση των δεδομένων διαμόρφωσης να προωθούνται και να απαντώνται από το etcd.

4.2 Μεταφορά των δεδομένων διαμόρφωσης στο etcd

Τα δεδομένα διαμόρφωσης της Ganeti συστοιχίας θα αποθηκεύονται πλέον στο etcd, και επομένως όλα τα αιτήματα εγγραφής και ανάγνωσης αυτών πρέπει να γίνονται μέσω etcd. Συγκεκριμένα, έγιναν αλλαγές στον ganeti-wconfd, τον δαίμονα του Ganeti που είναι υπεύθυνος για την εγγραφή των δεδομένων διαμόρφωσης. Πλέον, αντί να δημιουργεί τρία νήματα/εργάτες. που αποθηκεύουν τοπικά και διανέμουν τα δεδομένα διαμόρφωσης, δημιουργεί μόνο ένα το οποίο γράφει τα δεδομένα διαμόρφωσης στο etcd. Αυτό γίνεται με αίτηση έγγραφης των κλειδιών config.data και ssconf στο etcd. Διανομή δεν χρειάζεται, αφού εγγραφή στο etcd ισούται με διανομή τουλάχιστον σε μία πλειοψηφία της συστοιχίας etcd.

Επιπλέον, όλες οι οντότητες που έκαναν ανάγνωση των δεδομένων διαμόρφωσης για συγκεκριμένες λειτουργίες, τροποποιήθηκαν ώστε η ανάγνωση αυτή να γίνεται μέσω etcd, με αίτηση ανάγνωσης προς τη συστοιχία. Οι αναγνώσεις από το etcd έχουν ορισθεί έτσι ώστε να αποφεύγονται αναγνώσεις παλιών δεδομένων. Τέλος, διάφορες οντότητες εκτελούσαν τοπικούς ελέγχους για την ύπαρξή τοπικών αρχείων με δεδομένα διαμόρφωσης. Αυτοί οι έλεγχοι τροποποιήθηκαν, αφού τοπικά αρχεία πλέον δεν υπάρχουν, ώστε οι οντότητες να λειτουργούν κανονικά.

4.3 Μηχανισμός αυτόματης μετάπτωσης κύριου κόμβου

Η τελευταία απαραίτητη οντότητα είναι ένας μηχανισμός που θα εντοπίζει σφάλματα κύριου κόμβου και θα εκκινεί την διαδικασία μετάπτωσης κύριου κόμβου σε έναν κα-

τάλληλο υποψήφιο κόμβο. Ο μηχανισμός αυτός πρέπει να διασφαλίζει ότι μόνο ένας κόμβος λειτουργεί ως κύριος και να διαχειρίζεται σωστά τις διαμερίσεις συστοιχίας ώστε να αποφεύγονται καταστάσεις split-brain.

Θα υλοποιήσουμε έναν νέο δαίμονα και θα τον εισάγουμε στην υπηρεσία του Ganeti. Ο δαίμονας αυτός, ganeti-mcd, εκκινείται μόνο στους υποψήφιους κόμβους από τον ganeti-etcd. Επιπλέον, θα χρησιμοποιήσουμε ένα κατανεμημένο κλείδωμα στο etcd, η οποία θα χρησιμοποιείται από όλους τους ganeti-mcd δαίμονες που τρέχουν σε διαφορετικούς κόμβους. Συγκεκριμένα, στην αρχή λειτουργίας ο ganeti-mcd προσπαθεί να αποκτήσει το καταναμημένο κλείδωμα. Εάν τα καταφέρει, είτε εκκινεί την υπηρεσία του κύριου κόμβου είτε εκτελεί μετάπτωση κύριου κόμβου αναλόγως με τον αν ήταν ο κύριος κόμβος. Στη συνέχεια, επαναλαμβάνει την εξής διαδικασία: ελέγχει αν η υπηρεσία λειτουργεί κύριου κόμβου λειτουργεί ορθά, αν ναι ανανεώνει το κατανεμημέν κλείδωμα, αν όχι σταματάει τη λειτουργία κύριου κόμβου και ελευθερώνει το κλείδωμα. Το κλείδωμα έχει τη μορφή κλειδιού στο etcd το οποίο γράφεται με τη Time To Live ιδιότητα. Επομένως, εάν το κλειδί αυτό δεν υπάρχει τότε ο πρώτος ganeti-mcd θα το δημιουργήσει και θα το ανανεώνει ώστε κανείς άλλος ganeti-mcd να μην μπορεί να το αποκτήσει ώστε να ξεκινήσει λειτουργία κύριου κόμβου.

Η χρήση κατανεμημένου κλειδώματος μας εξασφαλίζει ότι μόνο ένας κόμβος θα εκκινήσει τη λειτουργία κύριου κόμβου. Επιπλέον σε περίπτωση διαμέρισης της συστοιχίας, η πολιτική πλειοψηφίας του etcd διασφαλίζει ότι το πολύ μία υποσυστοιχία θα μπορεί να γράψει στο etcd και να αποκτήσει το κλείδωμα ώστε να εκκινηθεί η λειτουργία κύριου κόμβου από έναν από τους κόμβους της υποσυστοιχίας. Επιπλέον, εάν ο κύριος κόμβος εντοπίσει σφάλμα λογισμικού στην υπηρεσία κύριου κόμβου τότε τερματίζει τη λειτουργία της και ελευθερώνει το κλείδωμα. Εάν παρουσιαστεί πρόβλημα υλικού στον κύριο κόμβο τότε δεν θα μπορέσει να ανανεώσει το κλείδωμα και ένας υποψήφιος κόμβος θα την αποκτήσει και θα γίνει κύριος κόμβος.

Η υλοποίηση αυτού του μηχανισμού εξασφαλίζει ότι σφάλματα κύριου κόμβου εντοπίζονται και διαχειρίζονται εγκαίρως. Επιπλέον διαχειρίζεται ορθά διαμερίσεις συστοιχίας και αποφεύγονται καταστάσεις split-brain.Η πιθανότητα απώλειας δεδομένων διαμόρφωσης εξαλείφεται καθώς ο κόμβος που αναλαμβάνει καθήκοντα κύριου κόμβου διαθέτει πάντα την πιο πρόσφατη έκδοση των δεδομένων μετάπτωσης μέσω του etcd.

5 Αξιολόγηση και μελλοντικές δυνατότητες

5.1 Αξιολόγηση

Η υλοποίηση μας επεκτείνει τη λειτουργία του Ganeti προσφέροντας αυτόματη μετάπτωση κύριου κόμβου. Παράλληλα εξασφαλίζει ορθή λειτουργία σε καταστάσεις διαμέρισης συστοιχίας και εξαλείφει τη πιθανότητα απώλειας δεδομένων διαμόρφωσης. Για να το πετύχει όμως αυτό εισάγει έναν σημαντικό περιορισμό, ο οποίος είναι όμως αναγκαίος για την ορθή λειτουργία της αυτόματης μετάπτωσης κύριου κόμβου. Ο περιορισμός, που πηγάζει από τη πολιτική απαρτίας του αλγόριθμου Raft που χρησιμοποιεί ο etcd, είναι: ”Ανά πάσα στιγμή μια πλειονότητα των υποψήφιων κόμβων πρέπει να είναι ενεργή και με υγιής επικοινωνία μεταξύ τους”. Εάν αυτή η συνθήκη δεν ισχύει τότε η υπηρεσία Ganeti σταματάει να λειτουργεί εώς ότου ικανοποιηθεί η συνθήκη

Επομένως, η υλοποίηση μας αυξάνει τη διαθεσιμότητα της υπηρεσίας Ganeti προσφέροντας γρήγορη,ασφαλής και αυτόματη μετάπτωση κύριου κόμβου, αλλά περιορίζει τα σενάρια στα οποία η υπηρεσία μπορεί να λειτουργήσει. Σε μία συστοιχία τριών κόμβων, η υλοποίηση μας εντοπίζει και επιλύει σφάλματα κύριου κόμβου εντός 40 δευτερολέπτων. Επιπλέον, η απόδοση της υλοποίησης μας διαφέρει από την απόδοση του συστήματος Ganeti, λόγω της αλλαγής στην αποθήκευση και διανομή των δεδομένων διαμόρφωσης. Η σύγκριση των δύο αποδόσεων εξαρτάται από την απόδοση του δικτύου και των τοπικών μέσων αποθήκευσης της συστοιχίας.

5.2 Μελλοντικές δυνατότητες

Η υλοποίηση μας μεταφέρει τα δεδομένα διαμόρφωσης στο etcd. Ο κύριος κόμβος του Ganeti αποθηκεύει επιπλέον πληροφορίες για τις εντολές που εκτελεί, καθώς και την κατάσταση του. Αυτές η πληροφορίες διαμοιράζονται και στους υποψήφιους κόμβους. Μελλοντικά αυτές οι πληροφορίες μπορούν να μεταφερθούν στον etcd ώστε να μην υπάρχει πιθανή απώλεια.

Επιπλέον, προτείνουμε την αλλαγή της αρχιτεκτονικής της υπηρεσίας Ganeti. Συγκεκριμένα, προτείνουμε την δημιουργία μίας οντότητας, η οποία θα είναι εξολοκλήρου υπεύθυνη για την αποθήκευση και τη διανομή των δεδομένων διαμόρφωσης και όλα

τα αιτήματα εγγραφής/ανάγνωσης των δεδομένων θα προωθούνται σε αυτήν. Οι υπόλοιπες οντότητες δεν θα γνωρίζουν πως υλοποιούνται αυτές οι λειτουργίες και το σύστημα αποθήκευσης που χρησιμοποιείται. Η νέα οντότητα θα υποστηρίζει διάφορα συστήματα αποθήκευσης και ο διαχειριστής θα επιλέγει ποιο θα χρησιμοποιηθεί κατά της διάρκεια εγκατάστασης του Ganeti. Έτσι, ο διαχειριστής θα έχει τη δυνατότητα να επιλέξει το κατάλληλο σύστημα ανάλογα με τις προτεραιότητες και τους περιορισμούς που έχει. Επιπλέον, η υπηρεσία Ganeti θα μπορεί εύκολα να εισάγει νέες τεχνολογίες και συστήματα αποθήκευσης.

1 Introduction

1.1 Problem

The problem under investigation is extending Google's Ganeti[RK14] cluster manager for High-Availability[1] with an automated master failover procedure. Ganeti is a Infrastructure as a Service (IaaS) system that offers virtual machines as resources. Virtual machines are spread over multiple nodes that form a cluster. Responsible for managing this cluster is a single node, the master node. If a failure occurs on the master node, another node has to take over as the master node. This operation, known as master failover, in the current Ganeti implementation is executed manually by an administrator. Manual execution of a master failover induces several problems but various approaches to automate this process were deemed inadequate.

1.2 Incentives

In the modern era where web applications and cloud computing are ever-growing, the demand of IaaS systems is rapidly increasing. Customers prefer to facilitate their applications on existing services that take care of infrastructure details like physical computing resources, location, data partitioning, scaling, security, backup etc. Vendors try to optimize their systems to increase their customer base and maximize profit. IaaS systems may be configured differently depending on the features they offer and priori-

[1]High-availability clusters are groups of computers that support server applications that can be reliably utilized with a minimum amount of downtime.

tize. In any case, IaaS system developers implement various design strategies[vV14] to maximize the uptime of their system, which is an important criterion for customers.

1.3 Shortcomings

Ganeti currently has a manual master failover procedure. An administrator has to detect and resolve a master node failure, a process that adds a considerable amount of downtime and the element of human error. Various Ganeti contributors and co-developers have proposed different approaches to solve this issue. Some of the approaches suffered from potential configuration data loss as they did not ensure that a node initiating a master failover has an up-to-date copy of the configuration data. Other approaches did not handle cluster partitions correctly and allowed multiple nodes to run as master leading to a split-brain condition[2]. Therefore these proposals were deemed naive and erroneous and were not imported to Ganeti.

1.4 Objective

Our objective is to extend Ganeti for high-availability with an automated master failover mechanism. Our approach has to eliminate the possibility of configuration data loss during a master failover *and* correctly handle cluster partitions to avoid a split-brain condition.

1.5 Design and evaluation

We propose integrating Etcd[Devc], which is a distributed reliable key-value store commonly used for storing the most critical data of a distributed system, as the back-end storage system to Ganeti. Etcd will offer a single and consistent image of Ganeti's configuration data throughout all nodes. Furthermore Etcd's quorum policy will allow us to correctly handle and recover from cluster partitions, avoiding split-brain

[2] A split-brain condition is the result of a Cluster Partition, where servers cannot communicate and synchronize their data with each other. Data or availability inconsistencies originate from this state as each side holds a separate data set and may proceed to take over shared resources.

conditions. A mechanism will be implemented in order to detect master node failure, whether that is a hardware/software failure or a cluster partition, and initiate a master failover procedure on a master candidate. Our implementation suffers from an important constraint that emanates from Etcd's quorum policy. In order for our implementation to run and make progress at any given time, a majority of the master candidates has to be active and communication between them has to be healthy. While this constraint can be crucial, Etcd's quorum policy is necessary to avoid split-brain conditions, a major objective of our approach. On a 3-node Ganeti cluster, our implementation can automatically detect and recover from a master node failure within 40 seconds while ensuring the consistency of the configuration data.

2 Background

This chapter will provide the basic cluster theory and discuss some of the challenges raised in such systems. It will also discuss two cluster tools that will be used in our implementation, Ganeti and Etcd.

2.1 Introduction to Cluster Theory and Challenges

A cluster is a type of parallel/distributed processing system, which consists of a collection of stand-alone computers, commonly referred as nodes, that cooperatively work together as a single, integrated computing resource. The activities of the computer nodes are organized by a software layer that sits atop of the nodes and allows users to treat the cluster as a large cohesive computing unit [Bak01], called ”clustering middleware”. Usually, but not always, the nodes are connected through a fast local area network, in order to minimize the communication overhead. Computer clustering differs from other approaches, such as peer to peer or grid computing, due to a centralized management approach which offers a single system image concept instead of a more distributed one.

Computer clusters were not invented by any specific vendor, but were created by the need of customers, who could not fit all their work on a single computer or needed a backup. This goes back to early 1960's. The history of early computer clusters is tightly connected with the history of early networks, as one of the primary reasons to develop a network was to to link computer resources. Ever since their creation, the applicability and deployment of computer clusters has grown immensely and today it ranges from

small business clusters with a couple of nodes to some of the fastest supercomputers in the world.

Computer clusters are used for computation-intensive purposes, rather than handling IO-oriented operations such as web service or databases. When used this way, computer clusters offer [enga] [engb]:

- Cost efficiency: The cluster technique is cost effective for the amount of power and processing speed being produced. It is more efficient and much cheaper compared to other solutions like setting up mainframe computers.
- Processing speed: Multiple high-speed computers work together to provide unified processing, and thus faster processing overall.
- Improved network infrastructure: Different LAN topologies are implemented to form a computer cluster. These networks create a highly efficient and effective infrastructure that prevents bottlenecks.
- Flexibility: Unlike mainframe computers, computer clusters can be upgraded to enhance the existing specifications or add extra components to the system.
- High availability of resources: If any single component fails in a computer cluster, the other machines continue to provide uninterrupted processing. This redundancy is lacking in mainframe systems.

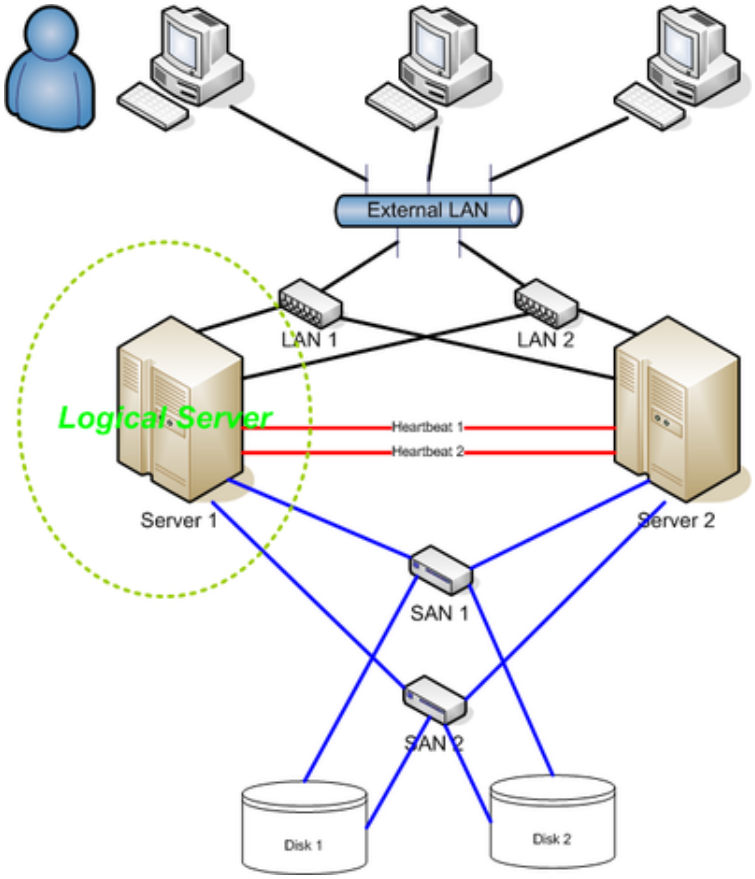
Computer clusters may be configured for different purposes, and thus they can be categorized to four major types of clusters, depending on the purpose they serve[Har99]:

- Storage
- High availability
- Load balancing
- High performance

Storage clusters provide a single,consistent file system image across all servers in the cluster, allowing servers to simultaneously read and write to a shared file system. By doing so, functions as backup and disaster recovery are simplified while the need for

redundant copies is eliminated. Moreover, administration of a storage cluster is easier by limiting the installation and patching of applications to a single file system.

Figure 2.1: *Example of a high availability cluster*



High availability clusters, also known as HA clusters or failover clusters, minimize the amount of downtime of server applications by eliminating single points of failure. This is achieved by using high-availability software that utilizes redundant computers. HA clustering detects software/hardware failures that occur on a node, and immediately restarts the application to a different node of the cluster without needing any administrative intervention, a functionality commonly known as failover. Maintaining data integrity between the two nodes is indispensable, in order to achieve failover without fomenting the application's behavior. Different architectures of HA clusters exist, depending on the number of active and passive nodes of the cluster[Ste01]. These are:

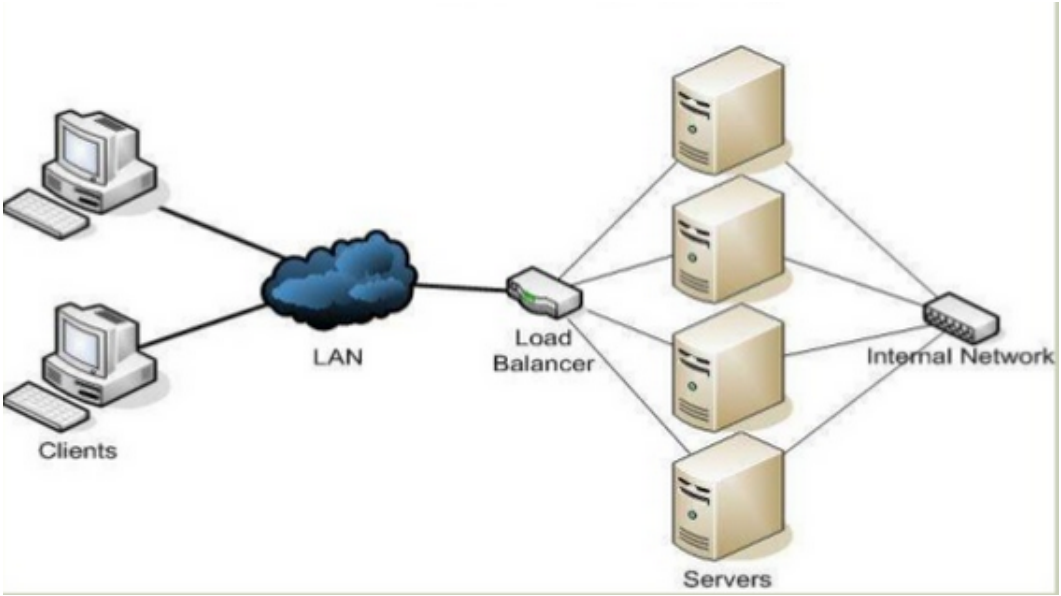
- active/active: All the nodes are active and traffic intended to the failed node can pass to any of the remaining nodes. Homogeneous software configuration is

needed for this architecture. Figure 2.1 shows an example of a active/active HA cluster architecture.

- active/passive: A single fully redundant node is available to each node, in case of a failure
- N+1: A single extra node is responsible for taking the role of any of N nodes should a failure occur
- N+M: M, more than one, extra nodes are responsible for taking the role of any of N nodes in case of a failure. This architecture increases redundancy but the multitude of M is a trade-off between cost and reliability requirements

Load balancing clusters are configurations were cluster nodes share workload in order to provide better performance. The algorithm that determines how to share the workload may differ depending on the nature of the service. For example, a web-server may use a simple round-robin method but a more sophisticated approach may be needed for a high-performance cluster used for scientific computations. If a node becomes inoperative, the load-balancing software detects the failure and redirects requests to other cluster nodes. A load-balancing cluster is shown in Figure 2.2.

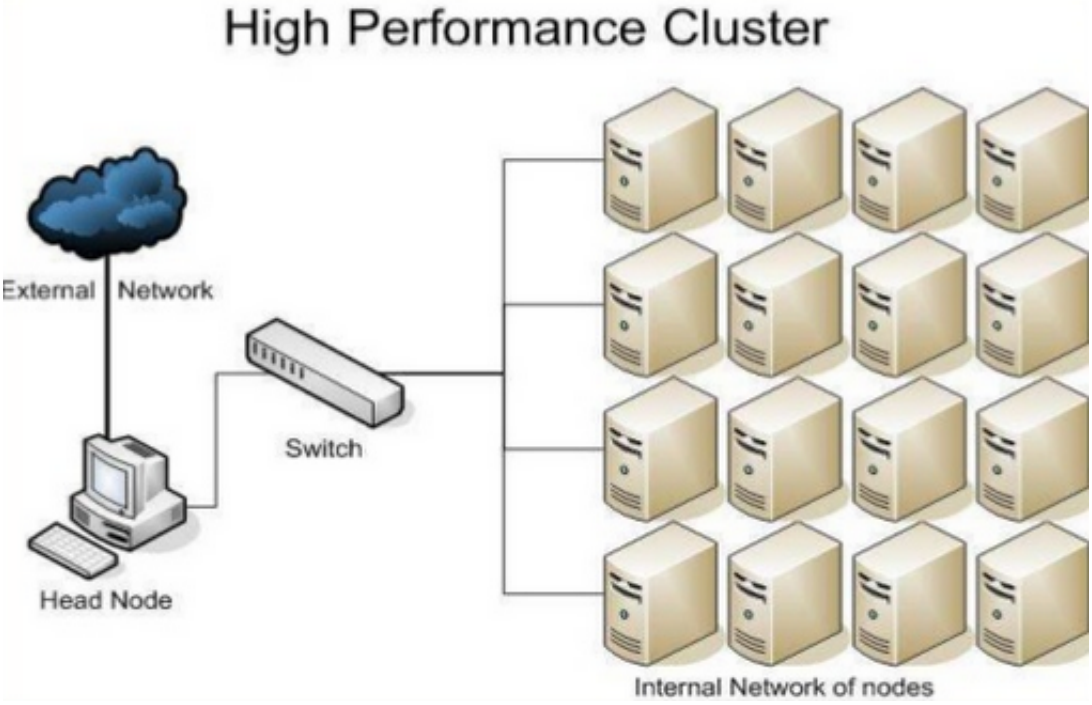
Figure 2.2: *Example of a load-balancing cluster*



High-performance clusters allow the application to work in parallel, by performing concurrent calculation on cluster nodes, therefore enhancing the performance of the

application. A usual high-performance cluster architecture, as shown in Figure 2.3, is when a single node splits and distributes the calculation workload between the node-workers and collects the results after the calculations are complete.

Figure 2.3: *Example of a high-performance cluster*



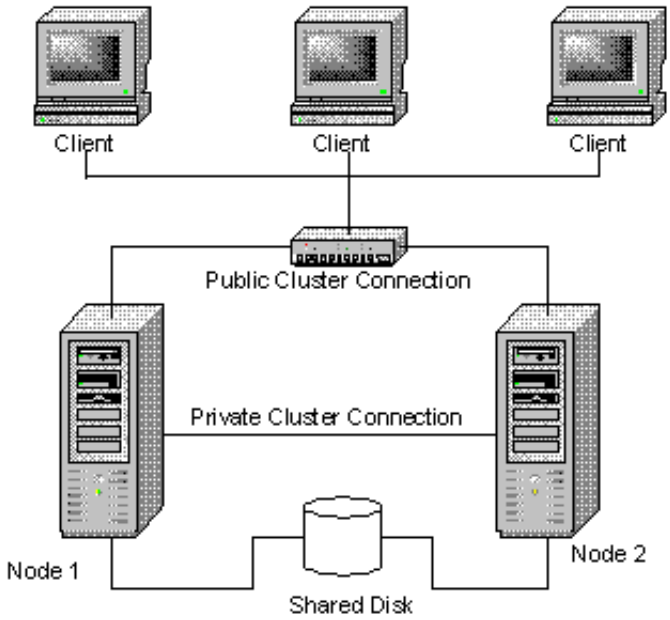
2.1.1 Split-brain

Depending on the type of cluster, there are a few delicate issues that need to be handled. One of them, and probably the most important, is split-brain. Split-brain is a condition that occurs as a result of a cluster partition. Cluster partition occur when there are communication failures between cluster nodes resulting to the cluster being divided into subclusters. Each subcluster consists of an arbitrary number of nodes that can communicate with each other but cannot communicate with any other node of the cluster that is not a part of this subcluster.

An example would be a two-node cluster consisting of node A and node B with a shared storage that allows read and write operations, such an architecture is shown in Figure 2.4. If A cannot communicate with B, and vice-versa, the cluster is in a split-brain condition. There are no means for node A to determine the state of node B. Node B could

be powered off or there is a network failure that blocks the communication between nodes A and B. Making an assumption about the state of node B , and therefore neglecting Dunn's Law[1], could lead to a variety of problematic scenarios. For example, node A could assume that node B is powered off and continue to serve read and write requests, and at the same time, node B could make the same assumption. This would lead to data corruption. Similar issues would arise for any shared resource between these nodes.

Figure 2.4: *A two-node cluster setup*



Since split-brain is the result of a cluster partition, an easy way to minimize split-brain occurrences is by eliminating a single point of failure in the communication system. This is done by configuring redundant and independent cluster communications paths so that loss of a single interface or path does not break communication between the nodes. Even with a redundant communication system, split-brain conditions can occur and thus should be handled properly.

There are various approaches that try to handle the split-brain problem. Davidson et al.[Ske85] classified them in two major classes:

- optimistic: The optimistic approach simply lets the partitioned nodes work as

[1] "What you don't know, you don't know - and you can't make it up". This law is attributed to former Raytheon vice president Bruce Dunn and it is highly applicable in the case of cluster failure where all that is observed is inability to communicate.

usual, which provides a higher level of availability in the cost of sacrificing correctness.

- pessimistic: The pessimistic approach limits the access of sub-partitions to shared resources in order to guarantee consistency. This approach sacrifices availability in exchange for consistency.

On the optimistic approach, once the problem has ended and the cluster partition has been resolved, automatic or manual reconciliation might be required in order to restore a consistent state to the cluster. A great example of this approach is Hazelcast, a key-value store system which values availability over consistency, that allows the system to keep running while a cluster partition has occurred and runs an automatic reconciliation after, in order to reach a consistent state.

On the other hand, on pessimistic approaches there are two basic concepts that take part, fencing and quorum. Fencing is the idea of putting a fence around a subcluster in order to deny any access to shared resources. In our previous example with a 2-node cluster setup, node A could fence node B and, after taking positive feedback for the fencing operation, continue to operate normally. Fencing in a way answers the question ”is the inability to communicate with node B caused by a network error or because node B is inactive?”, by forcefully setting node B unable to access shared resources, thus making him inactive. It is an excessive but effective way of ensuring that a consistent cluster state will be preserved. There are two classes of fencing:

- Node fencing is blocking a node from accessing any cluster resource, without knowing what resources it might be accessing. A common way of accomplishing this is by powering off or resetting the errand node. This technique is also called STONITH, which is an acronym standing for Shoot The Other Node In The Head.

- Resource fencing is a more elegant approach that can be used if there is knowledge about which resources the errand node uses or may use. In this case, a method can be used to block the node from using these specific resources, without powering it off. For example, if one has a disk that is accessed by a fiber channel, then one can communicate with the fiber channel switch in order to deny any access of the errand node.

In any case, an important aspect of good fencing techniques is that they are performed without the cooperation of the node being fenced off and give a positive confirmation that the fencing was done in order to continue safely. It is far better to rely on positive confirmation from a correctly operating fencing component, than to rely on errant cluster nodes you cannot communicate with to police themselves. In such a case, special hardware and software components are needed to perform fencing operations.

Another form of fencing, is self-fencing, where the node itself detects any inability to communicate with the rest of the cluster and proceeds by fencing itself. This can be done on a node level by resetting or on a resource level, by blocking access to shared resources. While this approach has the benefit of no extra hardware needs, as mentioned it is not prudent to rely on a malfunctioning component to fence itself.

While fencing can help a cluster to continue operating in a split brain condition, it not sufficient on its own. There is still the problem to decide which subcluster should be fenced off. On the two-node cluster approach, node A could try to fence off node B, and simultaneously node B could try to fence off node A. This could lead to an infinite reboot loop between the nodes. To solve this problem, the most-commonly method used is quorum.

Quorum is a method used to solve the mutual fencing dilemma [Cou01]. The main problem is to somehow select a single distinguished subcluster to carry on and fence off the rest. This must be done without communicating with any other subcluster, since this is the main problem. The most classic solution to selecting a single subcluster is a majority vote. In case of a cluster partition, every subcluster proceeds by counting the members that are part of it. If the majority of the cluster nodes are part of this subcluster, then it proceeds by fencing off the rest of the cluster nodes and continue normal cluster operation. Of course, there is the possibility that none of the subclusters reach a majority leading to a frozen state for the cluster, as no part of it can continue normal operation. Since, in real systems the most common cluster-partition splits the cluster in 2 subclusters most clusters consist of an odd number of nodes that will lead to a successful majority vote.

Unfortunately a majority vote approach is not eligible for a two-node cluster, which is a common cluster size, especially for failover clusters. In this case, there are a variety of other methods that use a third party arbitrator who selects what node to fence off.

This arbitrator can be either hardware or software. A hardware example would be a SCSI[2] reserve where both nodes try to reserve a disk partition available to both of them, and the SCSI reserve mechanism ensures that only one of them will succeed, and after doing so, proceeds to fence off the other node. A software example would be a quorum daemon, whose sole purpose is to arbitrate quorum disputes between cluster members. Such daemons are implemented in Linux-HA, HP and SUN systems, and operate more reliably and conveniently over geographical distances than the hardware solutions.

Using both fencing and quorum methods is highly recommended in order to successfully handle split-brain conditions.

2.1.2 CAP theorem

While referring to possible solutions to a split-brain condition, we encountered a case where the cluster designer had to sacrifice consistency over availability or vice versa. This is a common design dilemma when designing a clusters' behavior and is a effect of the CAP theorem.

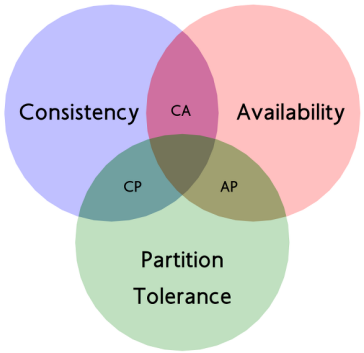
The CAP theorem[Bro][Tha], also named Brewer's Theorem, states that any networked shared-data system can have at most two of three desirable properties. These properties, as described by Seth Gilbert and Nancy Lynch[Gil02], are:

- **Consistency (C)** : Atomic, or linearizable, consistency is the condition expected by most web services today. Under this consistency guarantee, there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time.

- **Availability (A)**: For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response. That is, any algorithm used by the service must eventually terminate. When

[2]The Small Computer System Interface (SCSI) is a set of parallel interface standards developed by the American National Standards Institute (ANSI) for attaching printers, disk drives, scanners and other peripherals to computers.

Figure 2.5: *CAP theorem*



qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

- **Partition tolerance (P)** : The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes. Any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.

However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off. The ”two of three” formulation is misleading because it tends to oversimplify the tensions among properties. CAP, a visual conception is shown in Figure 2.5, prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare. Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its perceived limitations.

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing both sides to update their state will cause the nodes to become inconsistent, thus forfeiting C[Bre12][Fri96]. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A[Gre]. Only when nodes communicate is it possible to preserve both consistency and avail-

ability, thereby forfeiting P. The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A.

As already mentioned the ”two of three” view is misleading on several fonts. First, partitions are rare, thus forfeiting C or A when the system is not partitioned is not necessary. Second, the choice between C and A can occur many times within the same system, and the choice between those two can change depending on the subsystem, the specific data or even the user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances. So, a design choice can be made not between perfect C or perfect A but between a certain degree of C and A.

Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available. Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system's invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that may limit some operations, and initiate a

recovery process to restore consistency and compensate for mistakes made during a partition.

Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recovery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant. For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. In general, there is no way to tell if the operation will actually violate the invariant, since the state of the other side is not knowable. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation.

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually violated invariants and what results were externalized, including the responses

sent to the user. The designer must solve two hard problems during recovery:

- the state on both sides must become consistent
- there must be compensation for the mistakes made during partition mode

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. The tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated, which in turn enables the designer to create a restoration strategy for each such invariant. Typically, the system discovers the violation during recovery and must implement any fix at that time.

Summarizing, there a lot of different types of clusters, depending on the service they provide. Since cluster partitions cannot be avoided on a real system, restrictions made by CAP have to be considered. A cluster designer has to optimize the degree of availability and consistency the cluster offers, while at the same time respecting any invariants and limitations the service may have. A lot of different strategies and techniques exist and are implemented on current systems in order to satisfy the restrictions different type of services introduce.

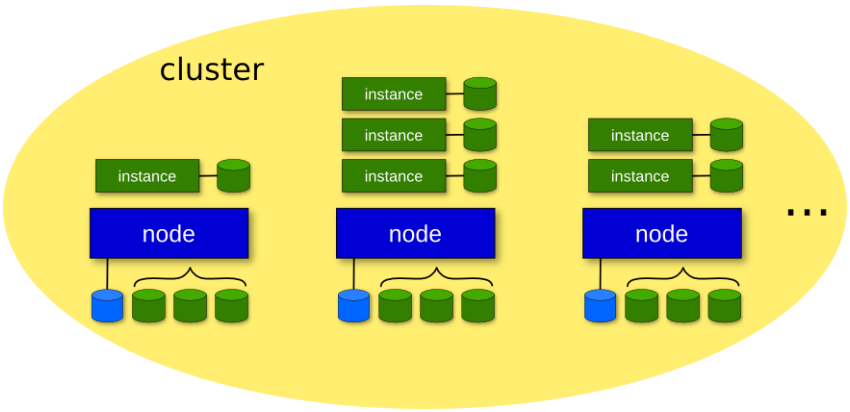
2.2 Ganeti

Ganeti is a virtual machine cluster management tool initially developed by Google and released as free and open-source software. An example of a Ganeti cluster is shown in Figure 2.6. Ganeti is designed to facilitate cluster management of virtual servers and to provide fast and simple recovery after physical failures using commodity software. It uses existing virtualization technologies such as Xen[3] or Kernel-based Virtual Machines. Once Ganeti is installed it assumes management of the virtual machines, also known as instances, by controlling:

- Disk creation management

[3]Xen Project is a hypervisor using a micro-kernel design, providing services that allow multiple computer operating systems to execute on the same computer hardware concurrently.

Figure 2.6: *Example of a ganeti cluster*



- Operating system installation for instances
- start-up,shutdown and fail-over between physical systems

Ganeti provides the following features for managed instances[Devr]

- Support for Xen virtualization
  - Support for Xen Paravirtualization[4] and Xen Full Virtualization[5] instances
  - Support for live migration
  - visual console to control instances
  - support for virtio[6] or emulated devices
- Support for KVM virtualization
  - Support for live migration
  - Support for fully virtualized instances
  - Support for semi-virtualized instances
  - support for virtio or emulated devices

[4]Paravirtualization is an efficient and lightweight virtualization technique, where paravirtualized guests require special kernel that is ported to run natively on Xen, so the guests are aware of the hypervisor and can run efficiently without emulation or virtual emulated hardware.

[5]Fully virtualized also known as HVM (Hardware Virtual Machine) guests require CPU virtualization extensions from the host CPU. Fully virtualized guests don't require special kernel. Fully virtualized guests are usually slower than paravirtualized guests, because of the required emulation.

[6]Virtio is a virtualization standard for network and disk device drivers where just the guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor. This enables guests to get high performance network and disk operations, and gives most of the performance benefits of paravirtualization.

- Cluster size of 1-150 physical nodes is recommended, but scales up to couple of thousands with special administrative action
- Disk management
  - Plain Logical Volume Manager volumes
  - Files
  - RAID1 functionality using DRBD[7] for quick recovery in case of physical system failure
  - support for third party storage solutions using External Storage Providers and shared filesystems
- Instance disk partitioning
- Export and import mechanism for backup purposes on migration between clusters
- Automated instance migration across clusters

Ganeti can be used in Infrastructure As A Service (IaaS) implementations, and with the functionality it provides, there are a few advantages in comparison with other IaaS solutions, such as OpenStack[8] and VMware[9]. These are:

- Ganeti's architecture is lightweighted and fairly easy to understand and deploy
- Scales really well for small/medium organization needs
- Highly customizable backend and built-in redundancy
- Requires a minimal administrative staff in order to maintain and upgrade

[7]DRBD is a distributed replicated storage system for the Linux platform. It is implemented as a kernel driver, several user-space management applications, and some shell scripts. DRBD is traditionally used in high availability (HA) computer clusters.

[8]OpenStack is a free and open-source software platform for cloud computing, whereby virtual servers and other resources are made available to customers. The software platform consists of interrelated components that control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data center

[9]VMware provides cloud computing and platform virtualization software and services. Where a hypervisor is installed on the physical server to allow for multiple virtual machines (VMs) to run on the same physical server and VM's migration between physical servers that share the same storage is simplified.

At the same time there are a few disadvantages such as:

- No Graphic's User Interface frontend is provided
- Application Programming Interface is not very cloud compatible and is not intended to be open for general users of the platform
- No official vendor support from Xen or KVM

Ganeti is written in Python and Haskell and its functionality is divided in several daemons, each one responsible for a specific task such as configuration management, instance creation, etc. Managing all aspects of the cluster is done by a Command Line Interface. The nodes of a Ganeti cluster have different roles and different functionality and responsibilities depending on them.

2.2.1 Ganeti cluster architecture and node roles

A node belonging to a Ganeti cluster can have one of the following roles at a given time[Devr]:

- Master node: is responsible for managing the cluster, accepting and carrying out requests about cluster configuration, instance creation etc. The requests received are divided to a number of different operations, knows as jobs, that are executed in order to achieve the desired functionality. Also, the master node is responsible for distributing cluster configuration to master candidates.
- Master candidate: is able to become the master node in case of a failure on the current master. Full cluster configuration and knowledge must be up-to-date in order to avoid erroneous situations
- Regular node: common node state where the purpose is to facilitate instances and accept requests from the master node. Cluster configuration knowledge is not required
- Drained node: state where the node is functioning normally but cannot receive new instances. This state is forced to nodes when they are evacuated for hardware repairs

- Offline node: there is a record in the cluster configuration about the node, but the master node will not communicate to this node for any request, and instances facilitated on it will be flagged as erroneous

The role of a Ganeti node determines the subset of Ganeti daemons that will run on the node. Beside the node role, there are other node flags that influence its behavior:

- Master capable: this flag denotes whether the node can become a master candidate. Setting this flag to 'no' is done when the node is impractical to become master due to networking or hardware constraints
- VM capable: this flag denotes whether the node can facilitate or not instances. This can be set to no when there are hardware constraints, or this node is intended to be master and by not hosting instances a faster processing of master requests may be achieved.

Changing a node's role can be done by an administrator executing specific commands provided by the Ganeti CLI on the master node. When determining a node's role hardware and network constraints should be considered as well as possible trade-offs. For example, increasing the master candidate pool[10] will increase the fault-tolerance of the cluster but also increase communication between the master node and the master candidates when distributing the cluster configuration.

2.2.2 Ganeti daemons

Ganeti divides its functionality to various daemons, each having a specific task to perform. These daemons are:

- ganeti-watcher
- ganeti-cleaner
- ganeti-mond
- ganeti-kvmd

[10]Master candidate pool is a cluster configuration number that determines the number of master candidates on the cluster

- ganeti-noded
- ganeti-confd
- ganeti-luxid
- ganeti-rapid
- ganeti-wconfd

The ganeti-watcher is a periodically run script that has two separate functions, one for the master node and another that runs on every node[Devp]. These are:

- Master operations: Will try to keep running all instances that are marker as up in the configuration file, by trying to start them a limited number of times. Moreover, it will archive old jobs that master has completed. Last, it will verify and repair any DRBD disks that are used in instances.
- Node operation: Depending on the role of the node, the ganeti-watcher is responsible for restarting a specific subset of Ganeti daemons that are appropriate for the current node, in a case of a software failure.

The ganeti-cleaner is a periodically run script, that runs on every node and removes old-files that are either master-specific or node-specific[Devh].

The Ganeti monitoring daemon, ganeti-mond, is responsible for running specific data collectors and provide the accumulated data through an HTTP interface. Data-collection is run periodically and the daemon listens to default port 1815 for TCP requests. The default port can change during the cluster initialization or during the start-up of the daemon by setting the -p flag[Devl].

The Ganeti kvm daemon, ganeti-kvmd, runs on all nodes when the KVM virtualization infrastructure is used for instances. The KVM daemon monitors KVM instances, through their Qemu Machine Protocol socket, by listening for particular shutdown,powerdown or stop events. These events determine if the instance was shutdown by a Ganeti user or due to an internal error and the result is communicated to Ganeti, which determines if instance migration is needed[Devj].

The Ganeti node daemon, ganeti-noded, runs on every Ganeti node and is responsible for the node functions in the Ganeti system. It listens to port 1811 for TCP requests and runs the corresponding action[Devm]. These requests can be:

- job queue replication requests, sent from the ganeti-luxid
- configuration replication requests, sent from ganeti-wconfd
- single jobs that change the state of the node, such as creating disks for instances,activating disks, starting/stopping instances, etc, sent from ganeti-wconfd.

The Ganeti configuration daemon, ganeti-confd, is automatically active on all master candidates, and its role is to provide a highly available and very fast way to query cluster configuration values. These values are obtained from the configuration file config.data that is distributed from the master node to all master candidates. Each candidate keeps a cached copy of the configuration in memory in order to minimize disk accesses. This cached copy is reloaded from disk automatically when it changes, with a rate limit of once per second. The provided information will be highly available, as in: a response will be available as long as a stable-enough connection between the client and at least one working master candidate is available, and its freshness will be best effort, the most recent reply from any of the master candidates will be returned, but it might still be older than the one available through the master node. The ganeti-confd daemons listens to a port 1814 for UDP, so each query can easily be sent to multiple servers, requests[Devi].

The Ganeti query daemon, ganeti-luxid, runs only on the master node and is responsible for replying to all the LUXI, an internal Ganeti protocol, queries. These queries include both queries about the configuration and the current live state of the Ganeti cluster and queries that actually change the status of the cluster by submitting jobs. Thus, ganeti-luxid is also responsible with managing the Ganeti job queue, which is a queue that contains all the jobs that were, are or will be done in order to materialize all the requests concerning the state of the cluster. When a job needs to be executed, the LuxiD will spawn a separate process tasked with the execution of that specific job, thus making it easier to terminate the job itself, if needed. When a job requires locks in order to atomically change the state of the cluster, LuxiD will request them from WConfD. In order to keep availability of the cluster in case of failure of the master

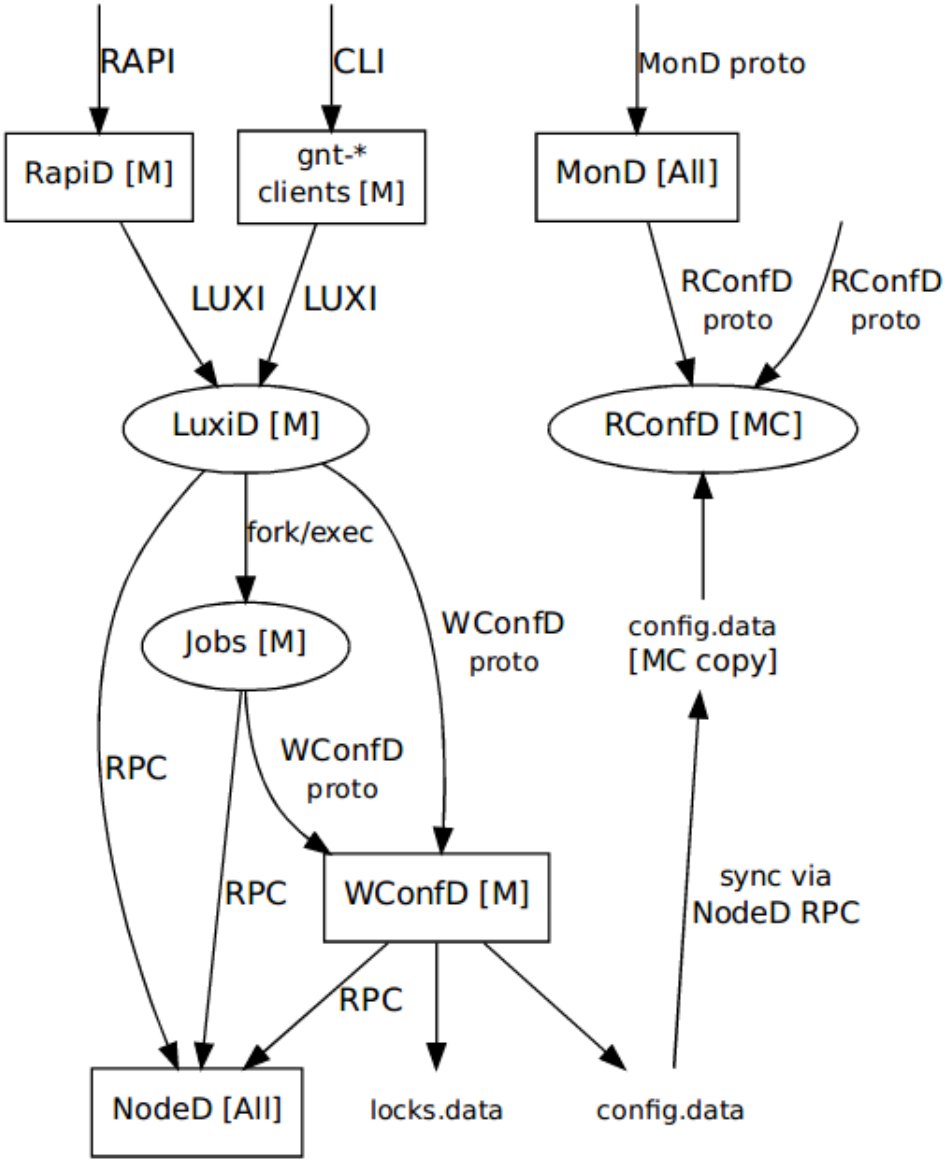
node, LuxiD will replicate the job queue to the other master candidates by sending job query replication requests to the ganeti-noded daemons that run on each master candidate[Devk].

The Ganeti remote API daemon, ganeti-rapid, is automatically started only on the master node and enables external tools to easily retrieve information about the cluster's state. The daemon listens to the "ganeti-rapi" port, default 5080, for requests and after ensuring the validity, in terms of abiding the Ganeti RAPI protocol and the authenticity of the user that sent the request, proceeds by forwarding the request to ganeti-luxid. Ganeti rapi daemon reads on start-up usernames and passwords from a specified file, default is /var/lib/ganeti/rapi/users, and changes to the file will be read automatically. SSL encryption is used by default. Ganeti-rapid listens to a port with the IP address given to the Ganeti cluster. This IP address is active on the master node. This allows the external tools to use that IP to communicate without knowing which node is the master nor having to change the IP each time a different Ganeti node takes over as master[Devn][Devo].

The Ganeti configuration writing daemon, ganeti-wconfd, runs only on the master node and is responsible for the management of the authoritative copy of the cluster configuration and is the only entity that can accept changes to it. All jobs that need to modify the configuration will do so by sending appropriate requests to this daemon. Ganeti-wconfd is also responsible for managing the locks, granting them to the jobs requesting them, and taking care of freeing them up if the jobs holding them crashed or are terminated before releasing them. Locks are used in order to ensure that the cluster configuration is changed atomically and is consistent. When the configuration is updated, it pushes the received changes to the other master candidates, via RPCs to the ganeti-noded daemon of each master candidate[Devq].

The basic Ganeti daemons and how they interact is shown in Figure 2.7.

Figure 2.7: *Interactions between the main ganeti daemons*



2.2.3 Ganeti CLI

Ganeti also offers a command line interface that enables the user to interact with the Ganeti cluster configuration by retrieving specific information or by changing the state of the cluster. These commands need to run on the master node, thus the user is a Ganeti cluster administrator. All of Ganeti's commands have a "gnt-" prefix and are classified to different groups depending on the functionality they provide. These groups are:

gnt-backup: Commands in this group are used for backing up instances and migrating them between different Ganeti clusters. The two basic commands of this group are ”gnt-backup export” and ”gnt-backup import” where the master node takes a snapshot of the current state of the instance and exports/imports it the node[Devs].

gnt-cluster: Commands is this group are used for cluster-wide administration of Ganeti cluster. Most important functionalities provided by this group are[Devt]:

- activating and de-activating the master ip
- initialize or destroy a Ganeti cluster
- show or modify various cluster parameters, such as enabled hypervisors used by instances, disk templates used for instance storage, etc
- initializing a master-failover
- verifying or distributing the cluster configuration
- verifying or renewing the encrypted certifications used for communications between the cluster nodes

gnt-debug: Various commands that test if various components of the Ganeti system, such as the jobqueue, the locking system, ganeti-wconfd are running correctly. These commands are used for debugging the Ganeti system[Devu].

gnt-group: These commands aim to simplify the administrators work by grouping different nodes to a single group. Running a command on a group of nodes translates to running the same command to every node of that group. This creates a logical unit that allows the administrator to modify node variables easier and faster. Node groups usually consist of nodes with similar hardware and network constraints[Devv].

gnt-instance: Commands is this group are used for adding, removing, modifying, migrating, starting, powering off or changing the disk management of an instance[Devw].

gnt-job: The gnt-job is used for getting the state of the cluster job queue and manipulating it. Commands that list the current job queue or give specific information about a job on the queue are implemented as well as commands that allow the administrator to cancel or watch a specified job[Devx].

gnt-network: It is used for the definition and administration of the network used for the Ganeti instance system. Network Interface Controllers, NIC, can be created and connect to various destinations using this command tool[Devy].

gnt-node: Gnt-node is used for managing the physical nodes of a Ganeti cluster. Most important functionalities provided by this group are[Gan]:

- adding and removing nodes from the Ganeti cluster
- modifying the parameters of a node
- evacuating a node from all instances that are facilitated on it
- managing the storage provided from a specific node to the instances that run on it
- get information and the nodes of the cluster and their state

gnt-os: The instances in the Ganeti cluster have various operating system. The gnt-os allows the administrator to see the list of the available operating systems for the instances, and to modify it by adding or removing an operating system or modify an existing one[Devz].

A short summary of the Ganeti system would be that a Ganeti cluster consist of nodes with different roles. The master node is responsible for modifying the cluster configuration and changing its status. In order to do so the ganeti-rapid, ganeti-luxid and ganeti-wconfd daemons, which provide functionalities that allow the modification of the configuration, start specifically on this node. The master node is also responsible for distributing the configuration to all master candidates. A master candidate node has an up-to-date copy of the cluster configuration and is eligible to become a master, should such a need arise. Moreover, on all master candidate nodes ganeti-confd daemon is running in order to provide a faster and highly available way to query for cluster information. A regular node is responsible for accepting and carrying out requests received from master, thus the ganeti-noded daemon which provides these functionalities runs on all regular nodes as well as all the above mentioned nodes. Instances are accommodated to all nodes that are flagged as vm capable. In order to view or change the cluster configuration and state, two tools are implemented. First, the Ganeti RAPI

which allows external tools to send such requests to a Ganeti cluster and second the internal command line interface that also creates such requests. There requests are forwarded to the ganeti-luxi daemon which creates jobs that perform all the actions needed for a specific requests. There jobs are queued to the Ganeti job queue and are executed one-by-one in order that depends on the submit ion time and the priority of each job. If a job intends to change the configuration of the cluster, ganeti-luxid has to request locks, that are used so that the configuration is atomically updated, from wconfd who is the solely responsible for managing a authoritative copy of the cluster configuration.

2.3 Raft and Etcd

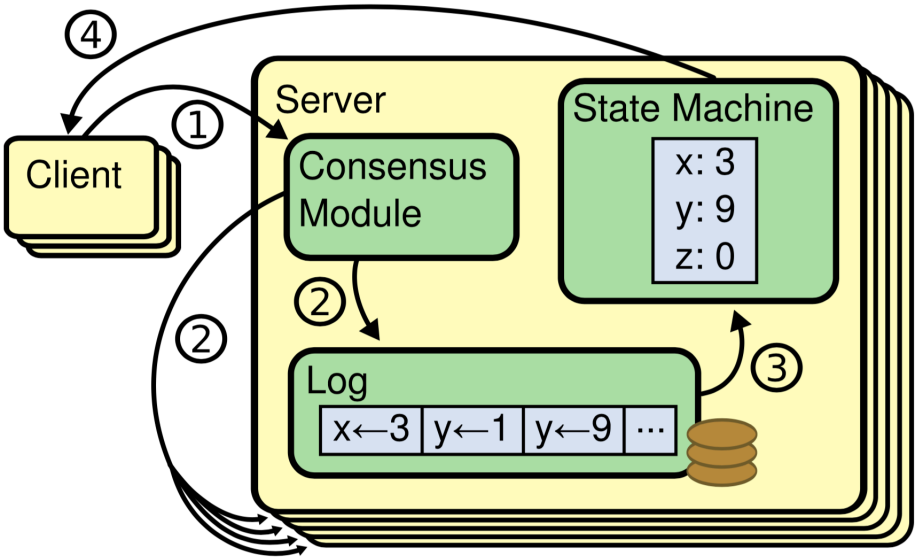
This section introduces Raft, a consensus algorithm that offers a generic way to distribute a state machine across a cluster of computing systems, ensuring that each node in the cluster agrees upon the same series of state transitions, and Etcd which is a distributed key-value store that uses Raft to ensure consistency.

2.3.1 Raft

Raft is a consensus algorithm for managing a replicated log. Raft was designed to be safe under all conditions and available under typical operating conditions, to provide a complete and practical foundation for system building in order to simplify the work of developers and to be easily understandable[Ong13]. The last two goals were set as Raft was designed as an alternate of Paxos, another consensus algorithm that is exceptionally difficult to understand as its full explanation is opaque and it does not provide good foundation for building practical implementations.

Consensus algorithms typically arise in the context of replicated state machines, where states machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines typically consist of a consensus module, a replicated log and a state machine as shown on Figure 2.8. Each server stores a log containing a series of commands that are executed in order by its state machine. Each log contains the same commands

Figure 2.8: *Replicated state machine architecture*

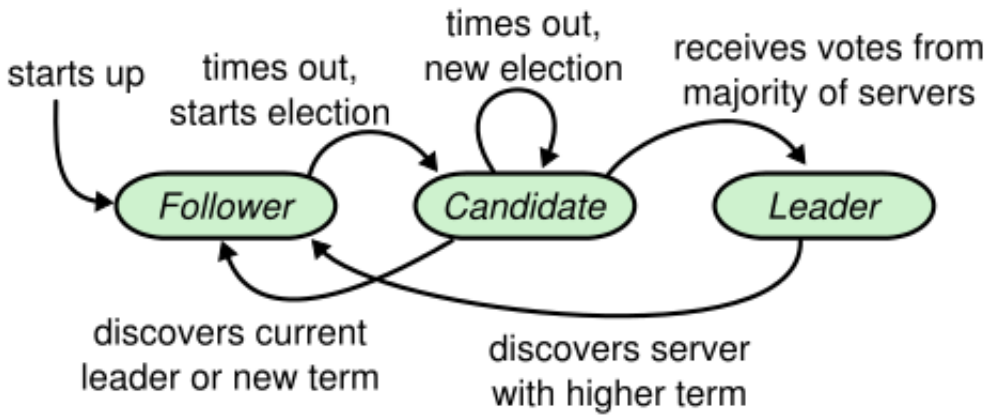


in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

Raft implements consensus by first electing a distinguished leader and giving him full responsibility for managing the replicated log, the form of which is described above. The leader accepts log entries from client, replicates them on the other servers and informs them when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log, as the leader can solely decide where to place the new entries in the log without consulting the other servers. Moreover the data flow is simple, from the leader to other servers. Given the leader approach, Raft decomposes the consensus problem into two relatively independent subproblems, an approach that also helps with the understandability of the algorithm. These subproblems are:

Figure 2.9: *Raft server roles and transitions between them*



- Leader election: a new leader must be chosen when the existing leader fails
- Log replication: the leader must accept log entries from clients and replicate them across the cluster

At any given time each server is in one of three states:

- leader: responsible for handling client requests and managing the replicated log
- candidate: a state during a leader election
- follower: passive role that simply responds to requests from leaders and candidates

During normal operation there is exactly one leader and all the other servers are followers. Raft divides time into terms of arbitrary length, which act as a logical clock and are increased monotonically over time. Each term begins with an election, where one or more candidates attempt to become leader. If one succeeds then it serves as leader for the rest of the term. If the election has no success due to a split vote then the term ends with no leader and a new one starts shortly. Each server stores a *current term* number that is exchanged is any communication with other servers. If the current term is smaller that the other's, then it is updated to the larger value. If the server is a candidate or leader state, it immediately reverts to follower. Raft's roles and transition between them is shown in Figure 2.9.

Leader election is triggered by a heartbeat[11] mechanism. All servers, at start up, assume the role of a follower and remain in this state as long as they receive valid heartbeats from a leader or a candidate. Leaders send periodic heartbeats to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the election timeout, then it assumes there is no viable leader and begins an election to choose a new leader.

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues a remote procedure call, RPC, requesting vote in parallel to each of the other servers in the cluster. A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term, a rule ensures that at most one candidate can win the election for a particular term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis. While waiting for votes, a candidate may receive an RPC from a server claiming to be the leader, and proceeds by reverting to follower if the term of the server is equal or larger than its own. Once a candidate wins an election, it becomes leader,and immediately sends heartbeat messages to all of the other servers to establish its authority and prevent new elections. Another possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election after a election timeout. The election timeout is chosen randomly between a fixed interval in order prevent split-votes from happening indefinitely.

Log replication is managed by the leader. When a client sends a request, the leader appends the command on its log and issues RPCs to all the other servers containing the command and the index and the term of the entry in its log that immediately precedes the new entry. A follower receives the RPC, compares the index and the term of the last entry with its own, if they match it proceeds with inserting the new entry in its log and sends a positive response to the leader, otherwise it sends an error respond to the leader, in order to inform him that its log is out-of-date. If such an error RPC is received by the leader, he proceeds by finding the latest log entry that matches with the log of that server,delete any entries in the follower's log after that point and send

[11] a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a computer system.

the follower all of the leader's entries after that point. The same procedure is done to every server after a leader election in order to minimize communications after. A log entry is committed, which means it is safe to apply to the state machines, when the leader receives a positive responses concerning this entry from a majority of the cluster. A positive response is then sent to the client that initiated the request. This log replication mechanism allows Raft to accept, replicate and apply new log entries as long as a majority of the servers are up.

Raft guarantees that each of these properties is true at all times[Ong13]:

- election safety: at most one leader can be elected in a given term
- Leader Append-Only: a leader never overwrites or deletes entries in its log, it only appends new entries
- Log Matching: if two logs contain an entry with the same index and term, the the logs are identical in all entries up through that index
- Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms
- State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index

These properties do not depend on timing[12].By ensuring these properties Raft achieves consensus of the log.Raft also define functionalities such as cluster membership live changes, log compaction and snapshots.

2.3.2 Etcd

Etcd is a distributed key-value store for the most critical data of a distributed system[Devc]. The name "etcd" originates from two ideas, the unix "etc" folder, which is a folder where configuration data for a single system is stored, and "d" from "distributed", thus "etcd". Etcd emphasizes on being:

[12]the system must not produce incorrect results just because some event happens more quickly or slowly than expected

- Simple: well-defined and user-friendly API
- Secure: automatic Transport Layer Security cryptographic protocols are used, with optional client certification authentication
- Fast: emphasis to fast write and read request time completion. Etcd is benchmarked with 10,000 writes/sec
- Reliable: Rigorous testing is done in order to ensure reliability and a faultless implementation of the Raft consensus algorithm

As a key-value store, etcd treats the data as a single opaque collection which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Key-value stores often use far less memory , than other store concepts such as Relational DataBases, which can lead to large performance gains in certain workloads.

Etcd stores data in a multiversion persistent key-value store, that preserves the previous version of a key-value pair when its value is superseded with new data. The key-value store is effectively immutable, its operations do not update the structure in-place, but instead always generates a new updated structure. All past versions of keys are still accessible and watchable after modification. To prevent the data store from growing indefinitely over time from maintaining old versions, the store may be compacted to shed the oldest versions of superseded data.

In order to achieve the above mentioned functionalities etcd internally stores the physical data as key-value pairs in a persistent b+tree[13]. Each revision of the store's state only contains the delta from its previous revision to be efficient. A single revision may correspond to multiple keys in the tree. The value of the key-value pair contains the modification from previous revision, thus one delta from previous revision. The b+tree is ordered by key in lexical byte-order. Ranged lookups over revision deltas are fast; this enables quickly finding modifications from one specific revision to another. By doing so, etcd provides a persistent, multi-version, concurrency-control data model that is a good fit to reliably store infrequently updated data, provide reliable watch

[13] A B+ tree, a data structure, is an n-array tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children

queries, expose previous versions of key-value pairs to support inexpensive snapshots and watch history events[Devf].

Etcd can be used as storage for metadata or important configuration data since it replicates all data within a single consistent replication group. Each modification of cluster state, which may change multiple keys, is assigned a global unique ID, called a revision in etcd, from a monotonically increasing counter for reasoning over ordering. Since there's only a single replication group, the modification request only needs to go through the Raft protocol to commit. By limiting consensus to one replication group, etcd gets distributed consistency with a simple protocol while achieving low latency and high throughput. On the other hand, having a single global replication group prohibits etcd from horizontally scaling since it lacks data sharding[14]. So, etcd is a very efficient for storing up to a few Gigabytes of data, for cases that surpass this magnitude of data another approach should be used, such as a newSQL database, that implements data sharding and allows horizontally scaling.

Etcd, since it offers primitives such as event watches, leases, elections, and distributed shared locks, can also be used for distributed coordination. In theory, it's possible to build these primitives atop any storage systems providing strong consistency but choosing etcd can help prevent operational headaches and save engineering effort.

Starting an etcd cluster requires that each member knows another in the cluster. Etcd can start with various ways, depending on whether information about all members and their IP addresses is available ahead of time or not[Deva]. These ways are:

- static: If there is knowledge of all the cluster members, their addresses and the size of the cluster before starting, an offline bootstrap configuration can be used, and each machine will be given the necessary variables in order to know on start-up all the cluster members and their addresses. Member changes are allowed after start-up by using the online cluster reconfiguration tool.

- etcd discovery: If there is no information about all the cluster members, their addresses and the size of the cluster before starting, the etcd discovery service can be used where a single discovery URL identifies a unique etcd cluster will be

[14] A database shard is a horizontal partition of data in a database or search engine. Each individual partition is referred to as a shard or database shard. Each shard is held on a separate database server instance, to spread load.

used by all members on start-up. This will cause each member to register itself with the custom etcd discovery service and begin the cluster once all machines have been registered[Devb].

- DNS discovery: Instead of etcd discovery a Domain Name Service SRV record can be used as a discovery mechanism for the members of the etcd cluster.

Etcd also offers an etcd proxy and an etcd gateway. While the etcd proxy can be used for improving performance and minimizing the load of the etcd cluster by coalescing multiple watch and lease requests, the etcd gateway as a simple TCP forward mechanism is only for simplifying access to the etcd cluster by multiple applications on the same machine[Deve][Devd].

Etcd is highly configurable through various command-line flags and environment variables. By setting these variables an administrator is able to:

- change the default security protocol TLS
- change the logging level in order to access more information about possible errors
- enable profiling of etcd and export specific metrics
- enable proxy
- configure cluster and member variables, such as snapshotting interval, port usage, etc

Furthermore, etcd has an authentication feature that specifies different users and roles. The most important user and role is root, that must be created before activating authentication. The root user is used for administrative purposes such as managing roles and ordinary users, modifying cluster membership, defragmenting the store and taking snapshots. The root user also has global read-write access and permission to update the cluster's authentication configuration. By adding users with different roles and privileges an administrator is able to allow limited access to the cluster to other users of the cluster.

An etcd cluster can suffer failures due to hardware or software malfunctions. The different kind of failures and how etcd is designed to tolerate them are cataloged here[Devg]:

- Minor followers failure: When fewer than half of the followers fail, the etcd cluster can still accept requests and make progress without any major disruption. However, any clients that access the etcd client through any of the failed members will lose connectivity, thus client libraries should hide these interruptions by automatically connecting to any other responding member leading to an increased load on these members.

- Leader failure: When a leader fails, the etcd cluster automatically elects a new leader. During the leader election the cluster cannot process any writes, and writes sent to the old leader but not yet committed may be lost. Write requests sent during the election are queued and will be processed after a new leader is elected. Read requests continue to function normally.

- Majority failure: If a majority of the members of the cluster fail, the etcd cluster cannot accept any write requests due to the inability to reach consensus. The cluster is in a ”frozen” state until a majority of the members becomes available again.

- Network partition: During a network partition an etcd cluster is partitioned to an arbitrary number of subclusters. If majority is achieved is any of the subclusters, then this subcluster can function normally and progress can be made. A leader election may be needed if the old leader is not a part of the subcluster. Once the network partition clears, the minority side automatically recognizes the leader of the majority side and recovers its state. If majority cannot be achieved in any of the subclusters, then the cluster is in a ”frozen” state and no progress can be made until the network partition clears.

Even though etcd can recover from most of the cluster failures, it is recommended to frequently back up the etcd cluster, by taking snapshots, in order to correctly recover from rare failures that etcd cannot handle.

Some use cases of etcd are:

- Container Linux by CoreOS: Application running on Container Linux gets automatic, zero-downtime Linux kernel updates. Container Linux uses locksmith to coordinate updates. locksmith implements a distributed semaphore over etcd to ensure only a subset of a cluster is rebooting at any given time.

- Kubernetes stores configuration data into etcd for service discovery and cluster management; etcd's consistency is crucial for correctly scheduling and operating services. The Kubernetes API server persists cluster state into etcd. It uses etcd's watch API to monitor the cluster and roll out critical configuration changes.

3 Design

In order to integrate Ganeti with an automated master failover procedure we have to check how specific functionalities are provided in the current implementation of Ganeti, and how those functionalities may be problematic in an automated scenario. These functionalities are:

- How Ganeti stores, changes and distributes the Ganeti cluster configuration
- How the master-failover functionality is implemented and what checks are done during its operation
- Is there a mechanism that triggers a master failover in the occurrence of a master failure

3.1 Ganeti configuration store and distribution policy

The entirety of the configuration of a Ganeti cluster is stored on a single file named config.data, usually in the directory */var/lib/ganeti*. From this file, specific data is extracted and grouped to various files with the prefix ssconf. These files are also stored at the same directory and contain information about a specific entity, such as the nodes of the cluster, the master candidates, etc. The config.data file is distributed to all master candidates and the ssconf files to all nodes of the cluster. As mentioned in section 2.2.2 the ganeti-wconfd daemon runs only on the master node and is responsible to change and distribute the cluster config. Requests to access the configuration of the

cluster are translated as jobs that are queued on the Ganeti job queue. These jobs are threads that execute different operations based on the request. In order to ensure the consistency of the configuration ganeti-wconfd has a single lock that any entity that wishes to access the configuration, has to acquire. This lock is shared by functions that read the config and acquired exclusively by functions that modify it.

Ganeti-wconfd also stores internally the configuration of the cluster as a data structure. This data structure is called *DaemonState* and is passed to threads acquiring the lock, to avoid unnecessary disk or memory accesses. Any changes to the configuration of the cluster are first done on this structure and then written to the config.data file. This is done while holding exclusively the lock and thus other job can not modify the configuration during this operation.

Ganeti-wconfd also on start-up creates three threads, named workers, each with a specific task:

- *saveConfigWorker*: thread that is responsible for writing the configuration to config.data file and also waking up the other two workers
- *distMCsWorker*: As its name implies, this thread is responsible for distributing the configuration data to the master candidates
- *distSSconfWorker*: This thread is responsible for distributing all the ssconf files to all the nodes of the cluster

Ganeti-wconfd also creates two more workers that are responsible for storing the lock state and the temporary reservations. These two workers don't affect the Ganeti configuration modification and distribution and thus we will not describe how they operate.

Communication between the workers and the main wconfd thread is done by haskell's *MVar*, a structure that is a mutable location. Each of the three mentioned workers has a different MVar that it watches for changes. The worker is inactive until a change on the MVar is noticed. When a notice occurs the worker wakes up and executes the procedure he is tasked for. Moreover another MVar is created for every worker that is used for passing the operation's result to the main wconfd thread.

SaveConfigWorker, when woken up by a change on its MVar, proceeds by atomically writing the new configuration state to config.data and wakes up distMCsWorker and

distSSconfWorker by writing the DaemonState on their MVar. SaveConfigWorker writes on the output MVar the return value of the atomic write operation that was executed. DistMcsWorker when woken up reads the DaemonState from its MVar and extracts the configuration data. Using the data it creates a list with the master candidates of the cluster and their IP addresses. It proceeds by creating a specific Remote Procedure Call(RPC) containing the configuration data and sending it to all the master candidates. Then it waits for the response of the master candidates, either a positive one or a timeout error. A timeout error is received if a candidate is inactive or there is a communication error between the nodes. A summary or the master candidates responses is then written to the output MVar. DistSSonfWorker operates in a similar fashion as the distMCsWorker. When woken up by a change on its MVAr, reads the DaemonState from input MVar, extracts the configuration data and from it creates a list containing all the cluster nodes and their IP addresses as well as a dictionary with every ssconf file and their value. A specific RPC containing the ssconf dictionary is afterwards sent to every node of the cluster, and a summary of their responses is written on the output MVar.

The ganeti-wconfd process grants the exclusive lock and a copy of the DaemonState, which contains the current configuration data, on a job and receives another DaemonState structure as a result. A check is done to the new DaemonState structure to determine:

- If there has been any modification to the configuration data by the job then a True value is returned, otherwise False. The value is stored on a variable named modified.
- If this modification has been done to some core values that need a different approach then True is returned otherwise False. The value is stored on a flag named *distSync*.

If the modified flag is set then the wconfd process proceeds by writing the new DaemonState on the *saveConfigWorker's MVar*. The saveConfigWorker is triggered and proceeds with a locally write of the new configuration and triggers the two other workers to distribute the config.data and the ssconf files as mentioned above. If the *distSync* flag is set then the main wconfd process waits for the responses summary of the distM-

CsWorker and distSSconfWorker before releasing the lock.

It is important to note that the wconfd process does not take any particular action when an error occurs in distributing the configuration, whether that is on config.data distribution to master candidates or the ssconf files distribution to all nodes. Specifically when such an error occurs when distributing the configuration the corresponding worker notes it to its output MVar. The wconfd process reads the output MVar of both distMCsWorker and distSSconfWorker and when an error is noted the only action taken is logging an error message in the ganeti-wconfd log. This can lead to problematic scenarios.

These scenarios have been tested and carried out on a experimental 3-node cluster. The master node could continue to operate and change the configuration of the cluster even when the other two nodes of the cluster, that were master candidates, were powered off. Ganeti also has push-only policy concerning the configuration, which means that only the master sends the configuration to the other nodes and a node cannot request for an update. Due to this policy, when a master candidate became active again, its configuration data will be out-of-date until the master sends new configuration to master candidates. This will occur only when the master receives a modification request and executes it.

The master does not implement any consistency or consensus check when distributing the configuration. This can potentially lead to out-of-date copies on master candidates. Therefore, it is really important to check the procedure where a master candidate takes over as master and what checks are done during this operation. Afterwards we will be able to determine if a master candidate with an outdated configuration can become master.

3.2 Ganeti master failover

Ganeti has implemented a master failover procedure on its command line interface. Specifically it is *gnt-node masterfailover* with a possible flag no-voting that will be analyzed below. This command has to be executed by a Ganeti cluster administrator that has local access to a master candidate node that is not the master.

A Ganeti master failover procedure incurs various checks. First, as mentioned, the node where master failover is initiated, reads its local configuration files and checks if it is a master candidate and not the master. If this is not the case an error message is returned and master failover is stopped. If the node meets these prerequisites then master failover continues with an important check which is a majority vote. The node reads from its local configuration the node list of the Ganeti cluster and proceeds by sending to all nodes a RPC request asking the node name of the Ganeti master, that each node has on its local configuration. The responses are gathered and grouped to three different groups:

- positive votes(P): The node initiating the master failover and the queried node agree on the current master node
- negative votes(N): The master node, as seen by the asked node, is different
- timed out(T): The node asked is either powered off or not reachable from the network, thus the request timed out

In order to continue the master failover procedure the node has to have a positive vote from the majority of the cluster. This can be formulated as $2 * P > P + N + T$, where $P + N + T$ is the entirety of the Ganeti cluster nodes. If this is not the case, then the master failover procedures is terminated with an appropriate error message.

The same check is executed when a node starts the master functionality without a master-failover. This is done when the cluster was inactive and became active by starting the Ganeti service on multiple nodes. The node that was master before the service became inactive will be prompted to start the master functionality and a vote like this will take part when starting ganeti-wconfd and ganeti-luxid. This check can be bypassed if the –no-voting flag is declared.

If the master failover procedure passes all the above mentioned checks, then the node proceeds by forcing the current master to deactivate its master-ip and stop the three Ganeti daemons that carry out the master node functionality. These daemons are ganeti-rapid, ganeti-luxid and ganeti-wconfd. Then the node proceeds by starting these daemons, activating master-ip and changing the master node on the cluster configuration which will be sent to all nodes, informing them that the master node has changed.

The majority voting check, does ensure that a majority of the cluster is active and sees the same master as this node it is not enough to ensure that the node, soon to be master, has an up-to-date copy of the configuration. A three node cluster example will exhibit such an scenario. Nodes A,B and C are all master candidates and node A is currently master. If node C becomes inactive and cluster configurations are made, nodes C configuration copy becomes out-of-date. If node A is powered off and node C becomes active then node C can become master should such an operation be initiated on it. That is because nodes B and C are active and both configurations of nodes B,C contain node A as master node. Therefore node C will pass all checks during a master failover even though its configuration data is out-of-date. This will lead to configuration data loss. All modifications that were executed while node C was inactive will be lost.

On the current Ganeti implementation there is no mechanism that triggers a master failover when a failure occurs on the current master node. This operation has to be executed by a Ganeti cluster administrator. The Ganeti administrator has to be aware of the Ganeti cluster state and if a master failure occurs, either by a node failure or a network partition, initiate a master failover procedure. In order to do so, the administrator has to:

- release the master-node resources that are used for master operation, such as the master-ip, specific daemons etc. This will ensure that if the problem originates from a cluster partition the cluster will not undergo a split brain condition
- choose a master candidate node to become the new master and ensure that this node has an up-to-date copy of the cluster configuration
- start the master failover operation on the chosen node

It is not always possible for the administrator to find a master candidate with an up-to-date copy of the configuration. As mentioned above a master can continue to operate and change the cluster configuration when all master candidates are offline, since no checks regarding master candidates's state are made when changing and distributing the configuration. Therefore, the administrator has to check the local configuration of the master and compare it with a master candidate's local configuration and possibly manually copy the configuration to the master candidate. This scenario is not possible if the master was totally destroyed and no access to its storage is possible, a case that

may seem extreme but should be taken into account. If this is the case then the administrator has to chose a master candidate as the next master without ensuring that it has an up-to-date copy of the configuration in order for the Ganeti cluster to continue operating.

Having an administrator execute a master failover induces several drawbacks. First, as mentioned above, there are extreme scenarios where the Ganeti cluster will lose some configuration data. Secondly, a considerable amount of time may be needed for the administrator to perform a master failover. The administrator first must realize there is a master failure on the cluster, then get access to the possibly destroyed master node and a master candidate to ensure configuration is up-to-date and then perform a master failover. This can take from several minutes to days. Finally, the procedure is prone to human errors.

3.3 Proposed changes

It has become apparent that the current implementation of Ganeti relies on an administrator to perform a master failover can be erroneous. Introducing a automated master failover procedure can eliminate the human error factor and increase the uptime of the Ganeti master service. To create such an automated procedure changing the method that Ganeti implements configuration distribution and master failover is necessary. Moreover, a mechanism must be implemented that will safely and securely detect any master node failures and initiate a master failover procedure on an appropriate master candidate.

A first approach to creating such a mechanism was to implement the Raft algorithm internally in Ganeti. The configuration distribution would follow the Raft log replication rules, making sure that a modification of the configuration is done only when there is a majority of the master candidates available and that they have accepted the change. If the majority was not available the master would return an error message informing the user. Moreover that master failover procedure would abide the leader election rules of the Raft algorithm. The serial number of the Ganeti configuration would be used as the epoch number of Raft. Votes would be granted to a candidate it the number is equal or larger that the current node's. Lastly, a heartbeat mechanism

would be introduced in order to initiate master failover operations. Such an approach would solve the shortcomings of the current Ganeti implementations and lead to an automated master failover functionality but it was deemed that an internal implementation of the Raft could be pesky.

Instead of implementing Raft on Ganeti, we decided that we will use a existing implementation of Raft as storage for the Ganeti configuration and a mechanism for triggering a master failover will use it. This implementation is *Etcd* that was thoroughly analyzed in Chapter 2. The main configuration file config.data and all ssconf files will be stored on etcd, and any entity that wishes to read or write them will use etcd operations. Moreover, any change to the Ganeti cluster membership will lead to a change to the etcd cluster membership. Last, new entities will be introduced which will handle the etcd cluster and implement the master failover trigger mechanism.

4 Implementation

As already mentioned, in order to extend the current ganeti functionalities with an automated master failover mechanism we will integrate etcd to ganeti. An etcd cluster will be created and managed by ganeti, The etcd cluster will have a similar membership as the ganeti cluster and will be used to provide a consistent replicated configuration of the ganeti cluster. Also, a distributed lock in the form of an etcd key will be used from the master failover trigger mechanism. Changes and additions that were made in this implementation can be grouped as:

- Changes to the current ganeti implementation in order to create and manage the etcd cluster
- A new entity that will be responsible for starting an etcd instance during the ganeti service start-up
- Changes to the current ganeti implementation in order for all write and read operations of the configuration to go through the etcd cluster
- A new entity that will serve as the master failover trigger mechanism

4.1 Managing the etcd cluster

Each node of the ganeti cluster will run an instance of etcd. It is important decide the members of the etcd cluster. A first approach is for every ganeti member to be a member of the etcd cluster. This approach is simple but does not depict correctly the

usage of etcd, which is to serve as storage for the ganeti configuration. Since ganeti master candidates are responsible for the configuration it deemed appropriate to set master candidates as members of the etcd cluster. Normal ganeti nodes will connect to the etcd cluster through an etcd gateway, which is a TCP proxy that will forward read queries from normal nodes to the etcd cluster. This approach is better as only master candidates should participate in log replication, majority votes and leader elections in the etcd cluster.

So every ganeti node will connect to the etcd cluster, either with a normal connection as a member or through a gateway, depending on whether it is a master candidate or not. This connection must take place during the ganeti service start-up. Specifically there has to be an etcd connection before ganeti-node and ganeti-conf daemons start, because these daemons need access to the ganeti configuration in order to start and operate. Therefore a new ganeti daemon, named ganeti-etcd, was implemented. Its responsibility is to create a correct etcd connection, normal or gateway, to the etcd cluster. This daemon was set to start before any other ganeti daemon on service start-up, in order to provide cluster configuration information to any other daemon that needs it. Ganeti-etcd is implemented in python and was added to the ganeti service initialization script.It follows ganeti conventions on how to start and stop daemons and can be used as any other ganeti daemon by ganeti administrator tools.

In order for ganeti-etcd to determine what kind of etcd connection to create, information about etcd members is needed. This information is the etcd members list which is equivalent to the ganeti master candidates list and is stored locally in a file, named masterCandidates. This file is distributed by the master node to all nodes if any changes to the membership occur and it is the only configuration that is stored locally. The information in this file may be out-of-date if the node was offline or unreachable during membership changes. In order for the ganeti-etcd to run properly only one member in the masterCandidates file has to be correct and active. The etcd membership information is necessary to connect correctly to the etcd cluster and storing it locally is unavoidable, since the node has no other configuration information that can be used to access the membership list.

Ganeti-etcd on start-up reads the masterCandidates file and creates a list. Afterwards it proceeds by sending an etcd membership query to all members of the list,using the

etcd command line tool *etcdctl*, as follows:

```
etcdctl --endpoints=local_member_list member list
```

This is needed because as mentioned information on the local masterCandidates file may be out-of-date and the above query will answer with the most up to date list. This query asks each member of the endpoints flag to answer with the member list of the etcd cluster that it is part of. Ganeti-etcd receives a response that can be:

- up-to-date member list: a majority of the etcd members is active and thus a response was sent containing an up-to-date membership list of the etcd cluster.
- timeout: there is no majority online thus a timeout was received

In the first case, ganeti-etcd proceeds by refreshing its data stored on the masterCandidates file and then determines if this node is part of the etcd cluster, and thus an ganeti master candidate, or not. If it is not a member then it connects to etcd by creating an etcd gateway. This is done by running the command 4.1

```
etcd gateway start --endpoints=renewed_member_list
  --listen-addr=127.0.0.1:2379
```

After creating such a connection any request received at listen-addr will be forwarded to one of the members in the endpoints flag. Such requests will be read requests initialized from ganeti entities wishing to read part of the ganeti configuration.

If the node is a member than a normal connection to etcd has to be achieved. We have used the static discovery etcd protocol, which means that a member when connecting for the first time to the etcd cluster has to have knowledge about the members. etcd creates a data directory for each different etcd cluster that a node connects to, and this directory name and location can be set when connecting. We have a specific data directory and checking its existence will determine if the node connects for the first time to this etcd cluster. If the data directory does not exist then specific initialization information is extracted by the member list by ganeti-etcd. This information is then used to initialize correctly the etcd connection. Specifically for each member of the etcd cluster, its IP address and the port that it listens for etcd peer communication has

to be determined. The IP address is found by using the host command line tool which is a simple Domain Name System look-up utility and the peer port is set to default port 2380 in every node. Then ganeti-etcd sets two environment variables, ETCD_-INITIAL_CLUSTER with a list of each member's IP and port and ETCD_INITIAL_-CLUSTER_STATE to "existing" since the etcd cluster already exists. After setting these variables ganeti-etcd proceeds with creating an etcd connection by running the code 4.1.

```
etcd --name=${node_name}
  --initial-advertise-peer-urls=http://${node_ip}:2380
  --listen-peer-urls=http://${node_ip}:2380
  --listen-client-urls=http://${node_ip}:2379,http://127.0.0.1:2379
  --advertise-client-urls=http://${node_ip}:2379 --data-dir=${data_dir}
```

All the above flags are necessary in order for etcd to run properly. The variables node_-name and node_ip are found by DNS query and data_dir is a constant declared in the ganeti constants file, and it is same for all nodes. If the node has already connected to etcd before, thus the etcd data_dir already exists, ganeti-etcd runs the above command without setting the initial cluster flags. It is important to note here that these commands are executed within the ganeti-etcd, which is written in python, using the appropriate libraries, such as os.subcommand and others depending on the functionality that is needed.

The case where ganeti-etcd receives a timeout response is more interesting and calls for a more careful approach. By getting a timeout response ganeti-etcd cannot determine if the node is a master candidate, and thus an etcd cluster member, or not. The only safe conclusion that can be made is that currently there is no etcd majority active that would allow for a member list response. If the decision was to keep polling for the etcd member list then no majority would ever be active. This is because no node would connect as a member and thus etcd would freeze at the same state. So an implementation decision was made that if a timeout response is received then ganeti-etcd will proceed optimistically and connect as a member by executing the above mentioned command. If the node was a member then the connection is correct, if not then the node will not be accepted to the etcd cluster. After connecting, ganeti-etcd proceeds with polling for the member list in a similar way as it does in the start. By following this

optimistic approach majority at some point of time will be achieved and ganeti-etcd will receive a member list response. This response is then parsed in order to determine if the node correctly connected as a member or not. If it is a member then the existing etcd connection is maintained, otherwise the existing connection is terminated and a gateway connection is executed instead.

The behavior of ganeti-etcd can be better explained with a four node cluster example where node A,B,C are master candidates and thus members of the etcd cluster and node D has normal role. If all nodes are down and then nodes A, B and D start operation and after a short delay node C also. When ganeti-etcd on nodes A, B and D tries to get the member list, a timeout response will be received. If we did not follow the optimistic approach all nodes would keep trying to get the member list with no success. After the timeout response is received nodes A, B and D will try to connect as members to the etcd cluster. Nodes A and B will succeed because they have the necessary local information stored that verifies them as members and a majority of the etcd cluster will be active. Nodes' A and B connection to etcd as members is kept. Node D will have a faulty connection and by polling to the master candidates list for a member list, a response will be received when the etcd majority is active. Node D will then determine that it is not a member of the etcd and will drop to a gateway connection. Node C will connect later and as a majority is already active it will receive a response and will connect as a member and no polling will be done afterwards.

Ganeti-etcd after connecting as a member to the etcd cluster, knowing that it is a master candidate in the ganeti cluster, it will proceed by starting ganeti-mcd, a new ganeti daemon that is implemented as the master failover trigger mechanism that will be analyzed further on, and then terminating. If the node is not a master candidate then ganeti-etcd terminates without starting the ganeti-mcd daemon.

We have seen how ganeti-etcd creates the correct connection to the etcd cluster that will allow other ganeti entities to access it for read and write operations. It is also important to change critical ganeti operations that change the membership of the ganeti cluster or the roles of the nodes to also change the membership of the etcd cluster. This operations are:

ganeti cluster initialization is done by the CLI command *gnt-cluster init*. This command is executed locally by an administrator on a node and it creates a single node

ganeti cluster by initializing its configuration, setting it as master and starting the ganeti service which was disabled as this node was not a part of any cluster. During this operation a new single node etcd cluster has to be created. As the node might have been part of an old ganeti cluster when initializing the etcd cluster we have to make sure that there is no existing etcd data_dir. If such a directory exists it is removed and then we proceed by making a new etcd cluster and connecting to it. This operation is incorporated within the code that implements the ganeti cluster initialization. Furthermore, it is important to create and connect to the etcd cluster before the ganeti initialization operation tries to write the configuration because the configuration has to be written on the etcd cluster. After the configuration is written on etcd we terminate the etcd connection as the initialization procedure proceeds by starting the ganeti service, and thus ganeti-etcd will run and it will create a new connection. Also the master_candidate file is written with a single entry. The command to initialize and connect to the etcd cluster is the same that the ganeti-etcd runs4.1, with an empty etcd data_dir and no initialization flags set.

The **ganeti add and remove node** operations are executed on the master node by using the CLI command *gnt-node add/remove node_name*. It is important to alter these operations in order to also change the membership of the ganeti cluster accordingly.

The add member operation can be split in several parts. First the master node creates a ssh[1] connection to the node, the administrator provides the necessary credentials, and ssh certificates are created for future connections. The second step is to connect to the node and transfer the necessary configuration data files and start ganeti-node daemon which is needed for the next step. The third is that the master actually adds the node to the cluster by modifying the cluster configuration. The last step is that the master node connects to the note and starts the ganeti service. In order to alter this operation to our needs two basic changes were needed:

- The second step instead of transferring the configuration data, a temporary connection to the etcd cluster is made and the necessary configuration data will be accessed through it.

- After the third step is completed, we change the membership of the etcd cluster

[1]Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network

accordingly. The master node has to check if the node was successfully added and if the node is a master candidate. If this is the case then it proceeds by adding it as a member to the etcd cluster by executing 4.1, renews the master_candidates file and sends a copy to all nodes. If the node is not a master candidate there is no need for these actions. Then step four is executed, after terminating the temporary etcd connection created at step 2.

```
1 etcdctl member add node_name http://node_ip:2380
```

The remove member operation is implemented in a similar way. The master node connects via ssh to the designated node, stops the ganeti service and removes any configuration files. It then proceeds by removing the node from the ganeti cluster configuration. The changes made is that before connecting the master node determines if the designated node is a master candidate. If that is the case the when connecting to the node it removes etcd data directory. In any case the master_candidate local file is removed and no other file deletions are needed as there are no local copies of the configuration. After removing the node, if that node was a master candidate, the Master node also has to remove it from the etcd cluster. In order to do so the node's etcd ID is retrieved with a etcd member list query and the node is removed, by running 4.1

```
1 etcdctl member remove node_id
```

After removing the node from etcd master node has to refresh the master_candidates file on all nodes.

Membership changes are also needed when a ganeti node is promoted to master candidate or demoted from master candidate to normal. This operation can be done on the master node by the CLI command *gnt-node modify –master-candidate=yes|no node*. This operation is simple, the master node first changes the configuration and then connects via ssh to the node and starts the ganeti-conf daemon, if the node is promoted to master candidate, or stops the ganeti-conf daemon if the node is demoted to normal role. The changes added in order to also manage the etcd cluster are in each case different.

- master candidate →normal: the master node has to also stop the ganeti-etcd and

ganeti-mcd daemons and then proceed to change etcd membership in a similar way as when removing a node. The etcd data directory has to be removed and the master_candidates file needs to be refreshed on all nodes

- normal →master candidate: the master node has to stop the ganeti-etcd daemon and then proceed to change etcd membership in a similar way as when adding a master candidate role. the master_candidates file needs to be refreshed on all nodes.

By implementing ganeti-etcd and adding the above mentioned changes in ganeti ganeti is able to manage the etcd cluster membership and the correct etcd connection is created to all the cluster nodes. This setup has been tested on a three node experimental cluster.

4.2 Migrating ganeti configuration to etcd

The ganeti cluster configuration is stored in the config.data file. This file contains all the cluster and instance information and it is used by the ganeti-wconfd daemon that runs on the master node. From this file specific information is extracted and stored in separate ssconf files. Each file stores a specific piece of information, such as the node list, the master node, etc. The ssconf files are used by ganeti-noded daemon on all nodes in order to access necessary information to execute specific operations.

In section 3.1 it was mentioned that ganeti-wconfd to store and distribute the configuration it creates three thread-workers, each with a specific task. The first worker is responsible for storing locally the configuration in the config.data file, the second is responsible for distributing config.data to all master candidates and the third is responsible for creating a dictionary containing all ssconf files and their values and then distributing this dictionary to all the nodes. These three workers run sequentially. This approach is not necessary when the configuration is written on etcd.

Instead of having three different workers, a single worker is created which is responsible for writing the config.data and the ssconf files to etcd. No distribution is needed because writing to etcd automatically replicates the data to a majority of the cluster. The single worker, named saveConfig, waits on a Haskell MVar for input from the main

ganeti-wconfd process. This occurs when ganeti-wconfd wants to modify the configuration. Ganeti-wconfd writes the modified data structure that contains the configuration in saveConfig's MVar. SaveConfig is noticed and starts operating. It writes the configuration to the config.data key on etcd and then extracts all ssconf values from the configuration and writes them on etcd. The code, implemented in *Haskell* as is the rest of ganeti-wconfd, implementing the operation is shown at 4.2.

```
1  saveConfig :: FilePath -- ^ Path to the config file
2     -> FStat -- ^ The initial state of the config. file
3     -> IO ConfegState -- ^ An action to read the current config
4     -> ResultG (AsyncWorker (Any, DistributionTarget) ())
5  saveConfig path fstat cdRef =
6     lift . mkStatefulAsyncTask
7        EMERGENCY "Can't write the master configuration file" fstat
8     $ \oldstat (Any flush, _ ) -> do
9        cd <- liftBase (csConfigData `liftM` cdRef)
10       liftIO $ writeFileToEtcd "config.data" ( J.encodeStrict cd )
11       liftIO $ writeSSConfToEtcd $ mkSSConf cd
12
13 writeFileToEtcd :: String -> String -> IO ()
14 writeFileToEtcd key value = do
15    (exitCode, _ , stdErr) <- readProcessWithExitCode "etcdctl"
          ["set","--", key, value] ""
16    if exitCode == ExitSuccess
17       then logDebug $ "key" ++key++ "succesfully written to etcd"
18       else logError $ "unsuccessfully written key"++key++"to etcd"
19
20 writeSSConfToEtcd :: M.Map SSKey [String] -> IO ()
21 writeSSConfToEtcd ssconfs = do
22    writeSSC keys values $ length keys
23  where
24    keys = M.keys ssconfs
25    values = M.elems ssconfs
26    writeSSC [] [] 0 = logInfo "SSConf files written succesfully"
27    writeSSC [] [] _ = logInfo "SSConf files written unsuccesfully"
28    writeSSC (k:ks) (v:vs) n = do
29       (exitCode, _ , _) <- readProcessWithExitCode "etcdctl" ["set","--",
             keyToFilename "" k , unlines v] ""
30       if exitCode == ExitSuccess
```

```
31          then writeSSC ks vs $ pred n
32          else writeSSC ks vs
```

If saveConfig is successful then config.data and the ssconf files were successfully written on etcd and are available to read by any ganeti entity.

The config.data key is stored initially in etcd during the ganeti cluster initialization procedure with a simple *etcdctl set config.data "config_string"* command. Every time the ganeti-wconf daemon starts it initializes the internal configuration data structure that it uses by reading the config.data value from etcd. This, and all etcd reads in this implementation, are done with the quorum flag set. When the quorum flag is set etcd treats the read operation as a write, which translates that the request is forwarded to the etcd leader that responds with the most recent committed value of the key. This prevents stale reads and is necessary in order to avoid configuration data loss.

By adding this changes to ganeti all operations that access config.data are done through etcd. More changes are needed in order to migrate all ssconf file reads to etcd.

The current ganeti implementation has accumulated all ssconf operations into a single python class. This class, called *SimpleStore*, implements different methods, like *GetMasterNode, GetNodeList, GetMasterCandidates, GetMasterCandidatesIPList*, etc, in order to read different ssconf files. Any ganeti entity that wishes to read any ssconf file needs to create a new *SimpleStore* object and use its methods.

All of SimpleStore methods determine which ssconf file should they read and then proceed by calling the *ReadSsconfFile* method, that is implemented in SimpleStore. ReadSsconfFile reads the local ssconf file and returns a list of its contents. We changed ReadSsconfFile to read from etcd with a simple *etcdctl get "key"* operation, instead of a read from a local file. Again the etcd read operation is done with the quorum flag set in order to avoid stale reads. By doing so, all ganeti entities that need information stored in any ssconf file will get this information through etcd, providing the most up-to-date value.

More changes were needed in order to avoid several failures. Various ganeti daemons, such as ganeti-noded, on start-up run checks in order to determine if the nodes state is healthy and all necessary data is there. These checks include checking the existence of the config.data and ssconf files. Since this files are no longer locally stored the checks

would fail. Thus, we either removed the checks on these files or changed them into an etcd query that lists the etcd contents depending on the situation. Similar checks are done by ganeti verification or debugging operation such as *gnt-cluster verify*. These operations were also changed.

After all these changes, the migration of the ganeti cluster configuration to etcd is completed. On a three node experimental cluster we were able to determine, after running several benchmarks and multiple scenarios, that ganeti continues to operate normally without any unexpected failures of its components.

It is important to note here that by migrating ganeti cluster configuration to etcd write and read operations cannot be successful if there is no majority of the etcd cluster, equally of the master candidates, active and reachable. Thus, any procedure that uses such operations will fail and return an error message either reporting the inability to either read or write the configuration.

4.3 Automatic mechanism for master failover

The last part that is necessary for an automated master failover procedure in ganeti is a mechanism that safely and securely recognizes a master node failure and triggers the master failover procedure on a suitable master candidate node. The master node failure can be due to hardware or software error on the master node or due to a network partition. This functionality is achieved by introducing the ganeti master candidate daemon, ganeti-mcd. ganeti-mcd is implemented in python and its operation starts by ganeti-etcd only on master candidates nodes. ganeti-mcd is responsible for starting the ganeti master service on a master candidate node as well as triggering a master failover. This is achieved with a distributed lock on etcd.

The distributed lock has the form of a key value on etcd that is set with a Time To Live, TTL, parameter. The TTL parameter sets a specific number of seconds after which the key expires. Each instance of ganeti-mcd that runs on every master candidate tries to set this distributed lock, named *masterUp*, by using the etcd flag *prevExist* and setting a specific TTL. By setting the prevExist flag, an etcd write operation first checks if that key already exists on etcd, and if it does it returns the error code 105 which translates as: the write operation failed as the key already exists. If they key does not exist then the

write operation is successful. This allows the key masterUp to be used as a distributed lock, because only a single instance of ganeti-mcd will be able to write the key and all others will fail.

The TTL parameter is really important. It allows us to use the masterUp key as a liveliness indicator of the node that has acquired the lock. The ganeti-mcd instance that succeeds in writing the masterUp key continues to refresh its value and the TTL parameter, restraining all other ganeti-mcd instances to take the lock. We will now analyze how ganeti-mcd operates.

Ganeti-mcd starts by running some initiation commands. After that, it enters an infinite loop where he tries to acquire the lock. First, ganeti-mcd sleeps for READ_INTERVAL and then tries to acquire the lock. This operation4.3 may fail for several reasons. The connection to etcd is faulty, the etcd cluster has no majority active and reachable and thus the quorum read fails or the masterUp key already exists. In all these cases ganeti-mcd returns to the start of the loop and tries again to acquire the lock after sleeping for READ_INTERVAL. If the operation succeeds then it means that there is a majority of the etcd cluster active and the key did not exist previously.

```
curl -L http://127.0.0.1:2379/v2/keys/masterUp?prevExist -XPUT -d
    value=True -d lockTTL
```

If ganeti-mcd successfully acquires the lock, it proceeds by spawning a separate process using Process from python's multiprocessing module. The process is responsible for starting the ganeti master service on this node by running 4.3.

```
def becomeMaster():
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.DEBUG)
    result=utils.RunCmd(['gnt-cluster', 'getmaster'],timeout=10)
    if result.failed:
        logger.error('etcd cluster lost quorum before succesfully starting
            master')
        return
    masterNode=result.stdout[:-1]
    currentNode=netutils.GetHostname().name
    logger.info('currently on node ' + currentNode + 'and master_node is'
```

```
        + masterNode)
11    if masterNode == currentNode :
12        logger.info('starting master, since i am the master        in the
              up-to-date config')
13        if startMaster():
14            logger.error('error while starting master, stopping and
                  restarting')
15    else:
16        logger.info('starting master-failover')
17        result=utils.RunCmd(['gnt-cluster', 'master-failover'])
18        if result.failed:
19            logger.error('error while executing master-failover,  stdout &
                  stderr were:')
20            logger.error(result.stdout + ' ' + result.stderr)
21    return
22
23 def startMaster():
24    result = utils.RunCmd([pathutils.DAEMON_UTIL, "start-master"])
25    return result.failed
```

The process first checks the current master of the ganeti cluster. If the master is a different node then a master failover procedure is initiated. If it is the same node then the master service is started using the daemon-util tool. *daemon-util start-master* basically starts the three ganeti daemons that consist the ganeti master service. These daemons , as already mentioned, are ganeti-rapi, ganeti-wconfd and ganeti-luxid.

After creating the process, ganeti-mcd enters a second infinite loop. Inside this loop ganeti-mcd first runs a check of the ganeti master service state. This is achieved by checking if ganeti-rapid, ganeti-wconfd and ganeti-luxid are running and if the ganeti master IP is active. This checks also takes into consideration the status of the created process. Specifically a time window of one minute is given to the process in order to successfully start the master service. The value of this time window is determined by the maximum time needed for a master failover or a start-master procedure to terminate. This window depends on the system and may need to be configured in different systems that may operate in a slower or faster pace. During this time window ganeti-wconfd does not run the above mentioned check because it is possible that the process is still trying to start the master service. After the time window pass if the check is

not successful then ganeti-mcd proceeds by terminating any ganeti master daemon that is still active and release the master IP. Then it proceeds by falling back to the first infinite loop and tries to become master node again. This approach is a form of self-fencing. Ganeti-mcd realizes that the master service does not run properly, and decides to release all resources and fall down to master candidate state. Then the lock will eventually be released and another ganeti-mcd instance will try to start correctly the master service.

If the check is successful, the ganeti-mcd knows that the master service runs properly and proceeds by refreshing the masterUp lock. This allows this node to remain master of the ganeti cluster since no other instance of ganeti-mcd will be able to acquire the lock. The lock is refreshed by running4.3 command which is similar with the command that tries to acquire the lock, but the prevExist flag is not set.

```
1 curl -L http://127.0.0.1:2379/v2/keys/masterUp -XPUT -d value=True -d
    lockTTL
```

Ganeti-mcd checks the output of the command. This is really important because any error in this operation means the inability of the node to retain its master status. The refresh command may fail due to an etcd connection error or due to a majority failure. A majority failure is possible if a majority of the nodes is offline or if a network partition occurred. Ganeti-mcd cannot differentiate these cases and in any case proceeds by releasing the master service resources and stopping the appropriate daemons. Then it drop backs to the first infinite loop in order to acquire the lock. If the refresh operation is successful ganeti-mcd sleeps for WRITE_INTERVAL time and then goes back to the start of the second infinite loop in order to check the master service status and refresh the lock.

It is important to specify the values of the READ_INTERVAL. WRITE_INTERVAL and lockTTL. A first approach was for ganeti-mcd to try and start the ganeti master service without spawning a separate process. This meant that the above mentioned intervals had to be greater that the time that is needed for the ganeti master service to start, which is around 40 seconds in our system. This is because during this procedure any other ganeti-mcd should no be able to acquire the lock and then try to become master. We changed our approach by creating a separate process that is responsible for

starting the ganeti master service and having ganeti-mcd monitor this process. This allows us to lower the intervals substantially. Specifically, in our experimental cluster, we set READ_INTERVAL and lockTTL to 6 seconds and WRITE_INTERVAL to 4 seconds. WRITE_INTERVAL has to be lesser than the other two intervals in order to always keep the lock and avoid race conditions.

Our current implementation is able to recognize a master failure in a short period of time and trigger a master failover operation. In the worst case scenario $lockTTL + READ\_INTERVAL$[2] is needed before a master failover is triggered. This scenario is when the master node refreshes the lock for lockTTL, right after the ganeti master service crashes all all other ganeti-mcd successfully read the lock just before lockTTL expires. Then a READ_INTERVAL is needed before the see that the lock is free. In our implementation, with the intervals set to 6 seconds and 4 seconds, the worst case scenario is 12 seconds.

On our three node experimental cluster we were able to test our implementation. Various scenarios were executed and the behavior of our system was checked. A normal scenario is where nodes A,B,C start the ganeti service roughly at the same time. In a short amount of time the etcd cluster has majority and a single ganeti-mcd instance acquires the lock and starts the ganeti master service, presumably node A. Node A then is either hard resetted or a artificial network partition is created. The ganeti-mcd lock is not refreshed and after a short period of time node B or node C will acquire it and start the ganeti master service. This operation in our system completes in under 40 seconds. No configuration data loss occurs since etcd is used as storage.

[2]lockTTL and READ_INTERVAL can have the same value, which is the case in our implementation

5 Evaluation & Discussion

Infrastructure as a Service, IaaS, is a basic cloud-service model where providers offer computing infrastructure, virtual machines and other resources, as a service to subscribers. High-availability of systems providing such a service is crucial and imperative for competitiveness and profit maximizing. Google's Ganeti VM cluster manager is such a service. Ganeti's current master failover policy is for an administrator to detect a master failure and execute a master failover procedure manually. This leads to a considerable amount of downtime and inserts the possibility of human error. We were able to implement an automated master failover procedure by using etcd as the storage backend for Ganeti and implementing a master failover trigger mechanism. The automated master failover functionality we implemented ensures configuration data integrity and operates correctly under cluster partitions. On our three node experimental Ganeti cluster a master node failure triggers the automated master failover functionality which completes within 40 seconds.

The integration of etcd to Ganeti leads to some drawbacks. These drawbacks emanate from etcd due to the immanent constraints of the Raft algorithm.

Majority constraint

Our implementation has two constraints that have to be satisfied:

1. A majority of ganeti members is active and reachable

2. A majority of etcd members, equally ganeti master candidates, is active and reachable

The first constraint emanates from the voting procedure during a Ganeti master service start-up or a master failover. This constraint has to be met only during the start of a the Ganeti master service and can be bypassed by using the *–no-voting* flag, but may lead to some strange scenarios. For example, a cluster with a hundred members with only three of them as master candidates is partitioned to a 2-node subcluster with two master candidates and a 98-node subcluster with 1 master candidate. The 2-node subcluster will be able to start the Ganeti master service and continue to operate. The second constraint emanates from the nature of etcd, which needs a majority of its member active and reachable to make progress. This constraint has to be satisfied in any given time.

If a network partition occurs, the current master has to satisfy the second constraint, meaning he has to be member of a subcluster where a majority of the master candidates is reachable. If this constraint is not met, if a subcluster that satisfies both constraints exists then a master candidate that is part of this subcluster will automatically execute a master failover procedure and the Ganeti cluster can continue to operate. If such a subcluster does not exist then the Ganeti cluster will freeze.

The original Ganeti implementation only has to satisfy the first constraint during the Ganeti master service start-up or a master failover. After these procedures complete a network partition does not affect the master's state. In the original Ganeti implementation a master will continue to operate even when all other nodes are unreachable.

In summary, a network partition or multiple node failures may lead to a Ganeti cluster freeze in our implementation. The original Ganeti implementation does not suffer in these scenarios and can continue to operate normally. This may increase the downtime of our system but is a necessary feature in order to achieve our two main objectives: no configuration data loss during a master failover and avoid split-brain conditions. This is a trade-off between the two implementations. Our approach automatically handles a master node failure and minimizes the downtime of such a scenario, but introduces scenarios where it will not be able to operate.

Moreover, these two constraints and when they are applied can be complicated. This emanates from our design decision to use only master candidates as etcd members. Simpler approaches would lead to less complicated constraints. Such approaches could be:

- All Ganeti members will be master candidates and thus members of etcd. This approach would lead to a single constraint. If a majority of Ganeti members is active and reachable then Ganeti service will operate normally and make progress. While this approach simplifies the constraint, it also increases the etcd communication. Furthermore, a Ganeti node may have hardware or software constraints that prohibit it from being a master candidate.

- All Ganeti members are also etcd members but not necessarily master candidates. This approach simplifies the constraints. Progress can be made if a majority of the nodes and a single master candidate is active and reachable. Etcd communication will increase again but nodes with software or hardware constraints can have the normal role and not master candidate.

Erroneous membership changes

Specific etcd membership changes lead to erroneous situations. The most common one is when a single node etcd cluster adds a second node to the cluster. When adding the second member etcd will temporarily lose majority until the second member successfully connects. In our implementation, during the temporary loss of majority, ganeti-mcd will be unable to refresh the lock and therefore self-fence itself from master status and release all master resources. To bypass this problematic situation a *ganeti administrator* has to terminate the ganeti-mcd daemon on the master node before adding the second member. Furthermore the administrator has to add a temporary member to etcd. That member can be another node or a process that listens to a different port. After the membership change to Ganeti completes, the administrator has to restart ganeti-mcd on the node and remove the redundant member from etcd. A master-failover may occur because the second node may acquire the masterUp lock. This problematic scenario can be generalized when the etcd cluster consists of $2*k+1, k\mathbb{EN}$ nodes, only $k+1$ are online and a new master candidate node is added.

Since membership changes are rare and usually supervised by an administrator, this problematic behavior is not crucial and can be resolved.

Performance

By integrating etcd to Ganeti, any configuration write operation has to complete the Raft replication protocol. While at first this operation may seem slower than the orig-

inal distribution method, a closer look is needed in order to compare these two methods.

- Original distribution method: A configuration write operation splits in 3 sub-operations, as described in section 3.1. The total workload of an operation can be summed as a single local write and sending $MC+T, where MC = number of master candidates, T = number of nodes$, RPCs. If the write operation has to be executed synchronously, wconfd waits for the response of all of these RPCs. An RPC sent for replicating config.data to a master candidate leads to a single local write before a response is sent. An RPC sent for ssconf replication to any node leads to multiple, equal to the number of ssconf files, local writes before a response is sent. If the write operation is executed asynchronously wconfd does not wait for the response of the RPCs. Most operations are executed asynchronously.

- etcd replication: an etcd write operation in order to complete has a workload of:

  - RPC sent to every etcd member
  - response from a majority of the etcd members
  - local write of the key

  This process is executed two times, once for config.data and once for the ssconf files where an etcd batch request is used. Etcd has implemented a lot of optimizations in order to maximize the performance of its replication procedure.

These two operation highly depend on the network and storage performance. A comparison between these two approaches on a system may lead to different results depending on the subsystems used for networking and local storage.

A different distribution policy would be to store onlyconfig.data on etcd and each Ganeti entity extract the specified ssconf value from it. This approach would reduce the cost of the write operation by having a single write request instead of two.

Moreover, in our current implementation all configuration read operations are executed with the quorum flag set. This means that every read request is forwarded to the etcd leader and serialized. The leader checks its own log for the last committed entry

of this key and returns its value. The serialization of all requests on the leader of etcd may diminish the performance. This can be resolved by either using etcd proxies for read operations or remove the quorum flag from specific read operations where a stale read is acceptable.

6 Future Work & Conclusions

We have mentioned a few possible changes in our policy and our implementation in order to resolve specific problems or optimize the performance. In this section we purpose future additions to our implementation .

Jobqueue migration to etcd

On our implementation we considered config.data and the ssconf files as the entirety of the configuration data of a Ganeti cluster. In reality, another important part of Ganeti's configuration is the jobqueue. As mentioned in section 2.2 every request to the master node is converted to a job and queued to Ganeti's jobqueue in order to be executed. Ganeti-luxi daemon replicates the jobqueue, alongside with the state of the locks and the temporary reservations, to all master candidates by sending RPCs to them each time the state of the jobqueue changes. When a master failover occurs, the new master initializes the status of its jobqueue by copying the local jobqueue data he has received from the previous master node. Locks and temporary reservation data is taken into account during the initialization.

The distribution of the jobqueue to master candidates follows the same policy as config.data distribution. The master node sends a single RPC and does not wait for a response. Moreover, a distribution check is not executed in order to ensure that the master candidates successfully received the jobqueue data. This may lead data loss of the jobqueue during a master failover. A job that was queued and waiting to be executed may be lost if a master failover occurs and the new master does not have an up-to-date copy of the jobqueue.

Our current implementation follows this policy. A migration of the jobqueue to etcd

would eliminate the possibility of data loss of the jobqueue during a master failover but would significantly increase the load of the etcd cluster.

Change the architecture of the ganeti service and support different back-end storage systems

The current architecture of the Ganeti service allows different entities to access the configuration by executing write or read operations. We purpose that a Ganeti entity is introduced that will be responsible for answering all configuration access requests. All Ganeti entities that wish to access the configuration will have to do so by sending requests to the new entity and will be unaware of the backed storage system that is used.

This new entity will offer a front-end API for read and write operations and will listen to a specific port for requests. Different back-end storage systems will be supported, such as local storage, etcd or even databases. The Ganeti administrator will have to choose the storage system during installation of the Ganeti service. The right versions of write and read operations will be installed. For example, a write operation on a local filesystem storage system will correspond to a local write and distribution of the data to other nodes. If etcd is used as the storage system then a write operation will correspond to an etcd write operation and distribution will be handled by the etcd system.

Each different storage system will have to treat specific write operations different. For example a write operation that changes the membership of the cluster by adding a member. An etcd implementation will have to recognize this membership change and execute a membership change on the etcd cluster as well, but a local filesystem storage will just have to copy the configuration data to the new node. Moreover, different storage systems will have to introduce entities to enable the storage system to operate. For example, an etcd implementation will have to start a connection to the etcd cluster when the Ganeti service starts. A database implementation would have similar behavior. These extra entities have to be implemented and added to the Ganeti service initialization during the installation of Ganeti.

By changing the Ganeti architecture we enable Ganeti to be more configurable and meet different standards. A Ganeti administrator will be able to decide the storage system that will be used during installation. The decision may be different depending

on priorities and constraints that the service has. Moreover, new back-end storage systems can be easily imported to Ganeti by implementing a few different operations enabling Ganeti to keep up with new and different storage technologies.

Bibliography

[Bak01] Mark Baker, *Cluster computing white paper*.

[Bre12] Eric Brewer, *Cap twelve years later: How the rules have changed*, Computer Magazine (2012).

[Bro] Julian Browne, *Brewer's cap theorem*, `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`.

[Cou01] Jean Coulouris, George; Dollimore, *Distributed systems: concepts and design*, 3rd ed., Addison Wesley, 2001.

[Deva] Etcd Developers, *Etcd clustering guide*, `https://github.com/coreos/etcd/blob/master/Documentation/op-guide/clustering.md`.

[Devb] Etcd Developers, *Etcd discovery protocol*, `https://github.com/coreos/etcd/blob/master/Documentation/dev-internal/discovery_protocol.md`.

[Devc] Etcd Developers, *Etcd documentation*, `https://coreos.com/etcd/docs/latest/`.

[Devd] Etcd Developers, *Etcd gateway*, `https://github.com/coreos/etcd/blob/master/Documentation/dev-internal/gateway.md`.

[Deve] Etcd Developers, *Etcd proxy*, `https://github.com/coreos/etcd/blob/master/Documentation/dev-internal/grpc_proxy.md`.

[Devf] Etcd Developers, *Etcd storage documentation*, https://github.com/coreos/etcd/blob/master/Documentation/learning/data_model.md.

[Devg] Etcd Developers, *Undestanding failures*, https://github.com/coreos/etcd/blob/master/Documentation/dev-internal/failures.md.

[Devh] Ganeti Developers, *Ganeti-cleaner manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-cleaner.html.

[Devi] Ganeti Developers, *Ganeti-confd manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-confd.html.

[Devj] Ganeti Developers, *Ganeti-kvmd manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-kvmd.html.

[Devk] Ganeti Developers, *Ganeti-luxid manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-luxid.html.

[Devl] Ganeti Developers, *Ganeti-mond manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-mond.html.

[Devm] Ganeti Developers, *Ganeti-noded manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-noded.html.

[Devn] Ganeti Developers, *Ganeti-rapid manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-rapid.html.

[Devo] Ganeti Developers, *Ganeti remote api*, http://docs.ganeti.org/ganeti/current/html/rapi.html.

[Devp] Ganeti Developers, *Ganeti-watcher manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-watcher.html.

[Devq] Ganeti Developers, *Ganeti-wconfd manpage*, http://docs.ganeti.org/ganeti/current/html/man-ganeti-wconfd.html.

[Devr] Ganeti Developers, *Ganeti's documentation*, http://docs.ganeti.org/ganeti/current/html/index.html.

[Devs] Ganeti Developers, *Gnt-backup manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-backup.html.

[Devt] Ganeti Developers, *Gnt-cluster manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-cluster.html.

[Devu] Ganeti Developers, *Gnt-debug manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-debug.html.

[Devv] Ganeti Developers, *Gnt-group manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-group.html.

[Devw] Ganeti Developers, *Gnt-instance manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-instance.html.

[Devx] Ganeti Developers, *Gnt-job manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-job.html.

[Devy] Ganeti Developers, *Gnt-network manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-network.html.

[Devz] Ganeti Developers, *Gnt-os manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-os.html.

[enga] IBM engineers, *Ibm cluster system: Benefits*, http://web.archive.org/web/20160429022854/http://www-03.ibm.com/systems/clusters/benefits.html.

[engb] Microsoft enginners, *Evaluating the benefits of clustering*, http://web.archive.org/web/20160422092651/https://technet.microsoft.com/en-us/library/cc778629(v=ws.10).aspx.

[Fri96] Ken Friedman, Roy; Birman, *Trading consistency for availability in distributed systems*, 1st ed., Cornell University, 1996.

[Gan] GanetiDevelopers, *Gnt-node manpage*, http://docs.ganeti.org/ganeti/current/html/man-gnt-node.html.

[Gil02] Nancy Gilbert, Seth; Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT news (2002).

[Gre] Robert Greiner, *Cap theorem: Revisited*, http://robertgreiner.com/2014/08/cap-theorem-revisited/.

[Har99] Forrest M. Hargrove, William W.; Hoffman, *Cluster computing: Linux taken to the extreme*, Linux Magazine (1999).

[Ong13] John Ongaro, Diego; Ousterhout, *In search of an undestandable consensus algorithm*.

[RK14] Muskan Bansal Rakesh Kumar, Sonu Agarwal, *Open source virtualization management using ganeti platform*, National Conference on Emerging Technologies in Computer Engineering (2014).

[Ske85] Susan B. Davidson; Hector Garcia-Molina; Dale Skeen, *Consistency in a partitioned network: a survey*, ACM Computing Surveys (CSUR) (1985).

[Ste01] Thomas Sterling, *An introduction to pc clusters for high performance computing*, International Journal of High Performance Computing Application (2001).

[Tha] Royans Tharakn, *Brewers cap theorem on distributed systems*, http://www.royans.net/wp/2010/02/14/brewers-cap-theorem-on-distributed-systems/.

[vV14] Sander van Vugt, *Pro linux high availability clustering*, 1st ed., Apress, 2014.