Service Management in NoSQL Data Stores via Replica-group Reconfigurations

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Evdoxos Bekas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina

July 2017

Examining Committee:

- **Δημακόπουλος Βασίλειος**, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Μαγκούτης Κωνσταντίνος (Επιβλέπων)**, Επίκουρος Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Πιτουρά Ευαγγελία**, Καθηγήτρια, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Acknowledgements

I would like to thank my supervisor, Prof. Kostas Magoutis, for all the guidance advice and motivation he has provided throughout my time of research and writing of this thesis.

Furthermore, I would like to thank Prof. Vassilios V. Dimakopoulos and Prof. Evaggelia Pitoura for agreeing to participate in my examination committee and for their comments on my thesis draft.

Table of Contents

List of Figures

List of Tables

Abstract

Evdoxos Bekas, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, July 2017.
Service Management in NoSQL Data Stores via Replica-group Reconfigurations.
Advisor: Kostas Magoutis, Assistant Professor.

Internet-scale services increasingly rely on NoSQL data store technologies for scalable, highly available data persistence. Workload variations or imbalances in data-store resource utilization typically require adaptation actions that involve movement of data (e.g., migrating data between nodes) and thus involve high overhead. In this thesis we propose the use of low-cost adaptation actions based on targeted reconfigurations of replica groups as an effective way to offer rapid response to performance degradation (such as when a node experiences a temporary resource shortage). Such reconfigurations can be exercised and controlled by a management system that uses measurement-based analysis to ensure they are only applied when their benefit outweighs their cost. Such a management system may also leverage external cross-layer notifications from an underlying container management system (CMS) about resource-specific conditions that are expected to impact the data store. We evaluate a prototype implementation of the management system in the context of the RethinkDB data store on Google Container Engine with the Kubernetes CMS, and on a dedicated cluster, using the Yahoo Cloud Serving Benchmark. Our results demonstrate that low-cost data-store adaptation actions can improve overall system manageability, availability, and performance, and that events exposed by the CMS can be leveraged to drive proactive adaptation actions, improving overall quality of service.

Εκτεταμενη Περιληψη

Εύδοξος Μπέκας, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2017.
Διαχείριση κατανεμημένων αποθετηρίων δεδομένων τύπου NoSQL μέσω αναδιοργανώσεων ομάδων αντιγράφων.
Επιβλέπων: Κώστας Μαγκούτης, Επίκουρος Καθηγητής.

Οι συνεχώς αυξανόμενες ανάγκες των διαδικτυακών εφαρμογών για υψηλή απόδοση και διαθεσιμότητα στην αποθήκευση δεδομένων έχουν οδηγήσει στη δημιουργία μιας νέας κατηγορίας κατανεμημένων αποθετηρίων δεδομένων (data stores) γνωστή με τον όρο NoSQL. Τα NoSQL συστήματα είναι βελτιστοποιημένα ως προς την επεκτασιμότητα και την κλιμακωσιμότητα σε σύγκριση με τα συστήματα SQL (δηλαδή με τα DBMS). Σε περιπτώσεις όπου υπάρχουν μεταβολές στον φόρτο εργασίας ή δεν είναι εφικτή η δέσμευση των απαιτούμενων υπολογιστικών πόρων από τα αποθετήρια δεδομένων, απαιτείται να γίνουν ενέργειες προσαρμογής οι οποίες περιλαμβάνουν μετακίνηση δεδομένων από κόμβο σε κόμβο εισάγοντας υψηλό φόρτο στο σύστημα. Στην παρούσα εργασία προτείνουμε την χρήση ενεργειών προσαρμογής χαμηλού κόστους βασισμένες σε στοχευμένες αναδιοργανώσεις ομάδων αντιγράφων όταν ένας κόμβος αντιμετωπίζει μια προσωρινή υπερφόρτωση. Αναδιοργανώσεις αυτού του τύπου μπορούν να προγραμματιστούν και εκτελεστούν από ένα σύστημα διαχείρισης του αποθετηρίου δεδομένων, εκτιμώντας την αποτελεσματικότητά τους με βάση ανάλυση συστηματικών μετρήσεων, που υποδεικνύει πότε το πλεονέκτημα από τη χρήση του μηχανισμού υπερβαίνει το κόστος του. Ένα τέτοιο σύστημα διαχείρισης μπορεί επίσης να αντλήσει σημαντικά πλεονεκτήματα από τη χρήση διαστρωματικών (cross layer) ειδοποιήσεων από σύστημα διαχείρισης υποδομής για επικείμενα γεγονότα (όπως κατάρρευση ή υπερφόρτωση) που αφορούν πόρους του συστήματος και που εκτιμάται ότι θα επηρεάσουν την απόδοση και διαθεσιμότητα του αποθετηρίου. Στα πλαίσια της εργασίας αυτής υλο-

ποιούμε ένα τέτοιο σύστημα διαχείρισης με βάση το αποθετήριο RethinkDB και το σύστημα διαχείρισης υποδομής Kubernetes, στην υποδομή Google Container Engine και σε υποδομή τοπικού cluster, με φόρτο παραγώμενο από το Yahoo Cloud Serving Benchmark.

Τα πειράματα που εκτελέστηκαν έχουν ως σκοπό να δείξουν τα οφέλη μια τέτοιας αρχιτεκτονικής για την διαχείριση αναδιοργανώσεων ομάδων δεδομένων. Εξετάζονται οι περιπτώσεις όπου γίνεται προδραστική αναδιοργάνωση ενός RethinkDB cluster βασιζόμενη σε ειδοποίηση (notification) ότι κάποιος κόμβος επρόκειτο να τερματιστεί, καθώς και σε περιπτώσεις όπου ένας κόμβος βρίσκεται υπό συνθήκες υψηλού φόρτου που δεν οφείλονται στο αποθετήριο δεδομένων. Με αυτή την ανάλυση στοχεύουμε να προσδιορίσουμε εάν οι προδραστικές ενέργειες προσαρμοστικότητας, στην περίπτωση που προηγουμένως έχει ληφθεί ειδοποίηση για τον τερματισμό κάποιο κόμβου, μπορούν να βελτιώσουν την απόδοση σε σχέση με την περίπτωση οπού ο τερματισμός του κόμβου γίνει χωρίς προειδοποίηση.

Επιπρόσθετα, πραγματοποιήθηκε πειραματική ανάλυση συγκρίνοντας την επίδραση, βάση δοθέντων Service-Level Objectives (SLOs), περιόδων υψηλού φόρτου (hot-spots) διαφορετικής διάρκειας στην επίδοση, σε σχέση με την επίδραση των αναδιοργανώσεων ομάδων δεδομένων που πραγματοποιήθηκαν για την αποφυγή αυτών των hot-spot με βάσει τα ίδια SLOs. Το κόστος της αναδιοργάνωσης ομάδων αντιγράφων για την αποφυγή υψηλού φόρτου είναι μικρότερο σε σχέση με την περίπτωση όπου δεν εκτελείται καμία ενέργεια (αναδιοργάνωσης ομάδων αντιγράφων), ακόμα και για περιπτώσεις όπου η διάρκεια που επηρεάζεται ένας κόμβος από υψηλό φόρτο είναι αρκετά μικρή. Επίσης σύμφωνα με την ανάλυση που πραγματοποιήθηκε (θεωρητική και πειραματική) η αποτελεσματικότητα του μηχανισμού αναδιοργάνωσης ομάδων αντιγράφων μειώνεται δραστικά με την αύξηση των κόμβων που επηρεάζονται ταυτόχρονα από υψηλό φόρτο. Επομένως, είναι αποδοτικότερο –όταν είναι εφικτό– να προγραμματίζεται η ανάθεση εργασιών υψηλού φόρτου έτσι ώστε να ελαχιστοποιείται ο χρόνος που συμβαίνουν ταυτόχρονα.

Τα αποτελέσματα υποδεικνύουν ότι ο μηχανισμός προσαρμογής βασιζόμενος στις στοχευμένες αναδιοργανώσεις ομάδων αντιγράφων μπορεί να βελτιώσει σημαντικά την διαθεσιμότητα, διαχειρισιμότητα, και να συνεισφέρει στην τήρηση των στόχων απόδοσης των εφαρμογών. Ταυτόχρονα, οι ειδοποιήσεις από σύστημα διαχείρισης υποδομής για επικείμενα γεγονότα που αναμένεται να επηρεάσουν την απόδοση του αποθετηρίου επιτρέπουν την εφαρμογή προδραστικών ενεργειών δια-

χείρισης της ποιότητας των παρεχομένων υπηρεσιών.

Chapter 1

Introduction

The rapid growth of Internet-scale applications starting in the early 2000s created the need for new distributed data stores that offer superior scalability and elasticity compared to what was possible with traditional SQL servers, giving rise to a new class of data stores referred to as NoSQL systems [3, 4, 5]. Since the inception of early NoSQL systems, research on improving the quality of service offered by distributed NoSQL data stores has led to the incorporation of autonomic features such as automated addition of new nodes or removal of crashed nodes, load balancing, elasticity, etc. A challenge with currently available adaptation actions on data stores however is that they typically require data movement and are thus expensive: For example, changing the primary key of a large table is known to be a costly and time consuming operation [6]. Load balancing performed by moving data across nodes, either by migrating replicas or by moving records across shards, are both expensive operations [7]. Adapting to varying workload via elasticity actions is known to take a significant amount of time and resources [8].

In systems which use primary-backup replication with a "strong leader" [9, 10, 11, 12, 13, 14], write operations (and often reads as well) are satisfied by any majority of replicas, which always involves the leader. The leader (or primary) typically takes a higher load than follower replicas (or backups). A recent example of a lightweight

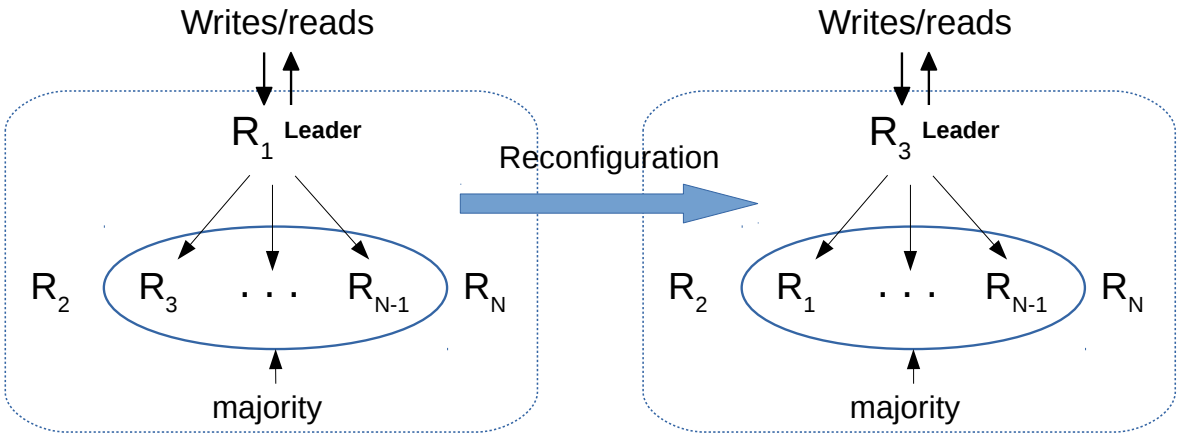


Figure 1.1: Reconfiguration on a primary-backup replication system

adaptation action that was shown to be beneficial in such systems, is to *reconfigure replica groups* as can be seen in Figure 1.1, by moving the leader away from nodes that are, or will soon be, heavily loaded [15]. In this way, a certain level of load balancing is feasible at low cost (the short availability lapse that such reconfigurations entail), occasionally leading to large benefits. A primary goal of the research in this thesis is to systematically explore the benefits possible through the use of this adaptation mechanism in achieving service-level objectives over NoSQL systems. While previous work demonstrated the benefits of the mechanism internally within a specific data store and to address a specific resource challenge [15], the work in this thesis demonstrates a more general use in the context of a *reconfiguration controller* that can address a wider-range of incidents affecting quality of service.

On the infrastructural front, the growing adoption of container technologies has brought container management systems [16, 17, 18, 19] (CMS) to the forefront of infrastructure management for large-scale data centers. Large cloud service providers such as Amazon, Google, and IBM are nowdays offering container clouds such as the Amazon EC2 Container Service [20], Google Container Engine [21] and IBM Bluemix Container Service [22]. NoSQL data stores are now available in containerized distributions that can be deployed in commercial and open-source CMSs. Just as over any tiered virtualization solution however, containerized data stores have limited information about the underlying infrastructure managed by a CMS, such as forthcoming disruptions affecting virtualized resources (decommissioning and replacement of a physical node, interference of resources across co-located workloads, etc). Our sec-

ond goal in this thesis is to enable and leverage a communication path between a CMS and a NoSQL data store, to notify the latter about infrastructure activities impacting its resources, as a way to drive proactive adaptation policies. We aim to show that the lightweight replica-group reconfiguration mechanism, when used proactively in response to infrastructure-level events provided by the underlying CMS, can improve data store *load balancing* and *availability*, without involving data movement actions.

Key to the enablement of such lightweight adaptation actions is explicit control over replica roles and their placement in data stores through a programmatic API. Such support is offered by systems such as RethinkDB [23] and MongoDB [24]. We chose to base the prototype of our architecture on the former due to its more straightforward cluster management API [25], but our implementation can be extended to cover the latter as well [13]. Such support is not ubiquitous in NoSQL data stores however, as replica roles and locations are not controllable in some data stores (e.g., using consistent hashing [26, 3, 27]) or are controlled internally.

1.1 Objectives

Our objectives in this thesis are the following: First, we aim to design an architecture for managing replica-group reconfigurations, at the core of which will be a reconfiguration controller that decides on appropriate reconfiguration actions when receiving events such as upcoming decommission or temporary load imbalance in the datastore cluster. We implement a prototype in the context of a specific NoSQL document store (RethinkDB) that is interoperable with a container management system (Kubernetes). The prototype can leverage events and associated upcalls by the container management system to drive cross-layer management policies that automatically map infrastructure events (such as new node added, node to be decommissioned) to datastore adaptation actions.

Second, we aim to perform an experimental evaluation demonstrating the benefits of such a management architecture in two scenarios on Google Container Engine deployments: Proactively reconfiguring a RethinkDB cluster on advance notification that a RethinkDB server is going to be decommissioned, and restoring service when a new server is available; and reconfiguring a RethinkDB cluster in the event of a hot-spot affecting performance of a RethinkDB server. This experimental evalua-

tion aims to determine whether proactive adaptivity actions in the case of previously announced downtime of a node can reduce the performance impact that would otherwise be experienced if the downtime was unscheduled (e.g., a crash), and whether load balancing actions via replica-group reconfigurations are effective in reducing the performance impact of hot-spots (overloaded nodes) on data store performance.

Finally, we aim to conduct an experimental analysis comparing the impact of performance hot-spots of different durations on service-level objectives (SLOs), to the impact of replica-group reconfigurations enacted to mask those hot-spots, on the same SLOs. This experimental analysis aims to answer the question of when does the cost of replica-group reconfigurations outweigh their benefits under different assumptions on hot-spot characteristics, especially their duration. For improved accuracy, experiments in this part of the thesis were planned to be carried out in a private computer cluster. Its results can be used to direct the process of deciding effective actions in the replica-group reconfiguration controller.

1.2 Thesis structure

The rest of this thesis is organized as follows: In Chapter 2 we describe background and related work, in Chapter 3 we outline the design of our service management architecture, and in Chapter 4 our prototype implementation. In Chapter 5 we present the evaluation of our prototype, highlighting key results, and finally in Chapter 6 we conclude.

Chapter 2

Background

2.1 Related work

In this section we provide background on NoSQL systems and their adaptation policies, container-management systems (CMS), along with a more in-depth description of two components with which our prototype interacts, RethinkDB and Kubernetes, relating to previous research along the way.

NoSQL data stores [4] are a new class of distributed data stores that appeared in the early 2000's [5] to satisfy the need of Internet applications for scalable storage of tabular data. They provide applications with a data abstraction that resembles traditional database tables offering semantics that lie between those offered by traditional databases (deemed too heavyweight to scale) and parallel file systems (lightweight but whose abstraction is too low-level to suit Internet applications). Adaptivity policies in data stores have been explored in the past. Cogo *et al*. [28] describe an approach to scale up stateful replicated services using replica migration mechanisms. They implemented an infrastructure called FITCH (*Fault- and Intrusion-Tolerant Cloud computing Hardpan*) to support dynamic adaptation of replicated services in cloud environments. They review several technical solutions regarding dynamic service adaptation and correlate them with motivations to adapt found in production systems, which they

want to satisfy with FITCH, such as increasing or reducing the number of computing instances, increasing and reducing the size or capacity of resources, moving replicas to different cloud providers or replacing faulty replicas. They show that it is possible to augment the dependability of such services through proactive recovery with minimal impact on their performance. Moreover, the use of FITCH allows both services to adapt to different workloads through scale-up/down and scale-out/in techniques.

Konstantinou *et al.* [7] describe a generic distributed module, DBalancer, that can be installed on top of a typical NoSQL data-store and provide a configurable load balancing mechanism. Balancing is performed by message exchanges and standard data movement operations supported by most modern NoSQL data-stores. While these approaches achieve adaptivity to a certain extent, they operate at the level of the application/NoSQL data store and cannot apply proactive policies based on advance information available at the infrastructure level. Cross-layer interaction between distributed storage systems and the underlying managed infrastructure has been explored by Papaioannou *et al.* [29], where co-location of virtual machines on the same physical host is detected by the middleware or exposed by the storage system (HDFS) to ensure that the placement of replicas avoids failure correlation to the extent possible. Wang *et al.* [30] propose cross-layer cooperation between VM host- and guest-layer schedulers for optimizing resource management and application performance.

In systems which use primary-backup replication with a "strong leader" [9, 10, 11, 12, 13, 14], write operations (and often reads as well) are satisfied by any majority of replicas, which always involves the leader. The leader (or primary) typically takes a higher load than follower replicas (or backups). Standard adaptation solutions available in stateless systems, such as steering load away from a temporarily overloaded server via load-balancing actions, are not easily applicable to data stores since this requires data migrations, a heavyweight and time-consuming activity. Replica-group reconfigurations, used in this thesis, is a lightweight alternative that is especially fit for temporary overload conditions in data stores. Previous work using replica group reconfigurations to mask the impact of underperforming primaries includes ACaZoo [15], a data store that triggers re-elections at replica groups with primaries on nodes that are about to engage in heavy compaction activities, typical in systems using log-structure merge (LSM) trees. By moving the primary role of a replica group away from a node that is about to either become overloaded or to crash, we ensure better

performance (as the primary will not block progress of the entire replica group) and availability. In this thesis we use similar replica-group reconfigurations when notified about infrastructure-level events by the CMS.

Borg [16, 17] was the first container-management system developed at Google at the time container support was becoming available in the Linux kernel. It aims to provide a platform for automating the deployment, scaling, and operations of application containers across clusters of hosts. Borg manages both long-running latency-sensitive user-facing services and CPU-hungry batch jobs. It takes advantage of container technology for better isolation and for sharing machines between these two types of applications, thus increasing resource utilization. Omega [18], also developed at Google, has similar goals to Borg. It stores the state of the cluster in a centralized Paxos-based transaction-oriented store accessed by the different parts of the cluster control plane (such as schedulers), using optimistic concurrency control to handle conflicts. Thus different schedulers and other peer components can simultaneously access the store, rather than going through a centralized master.

Kubernetes [19] is an open source container-cluster manager originally designed by Google and inspired by Borg and Omega, donated to the Cloud Native Computing Foundation [31]. We describe Kubernetes in more detail in Section 2.2.3. Google offers Kubernetes as the standard CMS in Google Container Engine (termed GKE) [21] within Google Cloud Platform. Another commercial CMS is Amazon's EC2 Container Service (ECS). ECS, like Kubernetes and Google's GKE, supports lifecycle hooks [32], namely upcall notifications of instance bootstrap or termination, an important cross-layer management primitive leveraged by our architecture.

Prior work on cluster management services includes Autopilot [33], an early approach to automate datacenter operations. Other approaches to cluster and datacenter management opting for a goal-oriented declarative style, have been a subject of research for some time [34]. Cross-layer management has previously been explored as way to break through the transparency enforced by multiple organizational layers in distributed systems [35, 36].

2.2 Systems used in this thesis

2.2.1 RethinkDB

RethinkDB [23] is an open-source document oriented, horizontally scalable NoSQL database. It stores JSON documents with dynamic schemas, and supports sharding (horizontal partitioning) and replication and has its own query language called *ReQL* [37] to express operations on JSON documents. RethinkDB is the first open-source, scalable JSON database built from the ground up for the realtime web. It inverts the traditional database architecture by exposing an a new access-model and instead of polling for changes is able to continuously push the updated query results to applications in realtime.

NoSQL schema

RethinkDB is a schema less database and it stores documents on logical containers which called *tables*. A table does not require any particular structure of it's contents but its generally beneficial for all documents it the same table to have the same structure, as it simplifies organization and management. RethinkDB documents are hierarchical, dynamically typed JSON objects consisting of key-value pairs and can be either a collection of name/value pairs or a ordered list of values. Each document is uniquely identified by a key within a *table*.

Realtime push architecture

RethinkDB features an option named *changefeed* [38] which is a build-in change notification system that can be used to simplify the development of real-time applications. Traditionally a client was able to be aware of a change in a database only by querying the database itself. In RethinkDB clients can subscribe to database changes and be notified automatically once there is any change without emerging a polling mechanism. RethinkDB realtime push architecture reduces dramatically the time and effort necessary to build realtime applications compared to traditional polling technique which is cost prohibitive in many cases. The *changefeed* option can be applied for a table, a document or a query and can be configured to throttle the delivery of information to reduce network traffic. For example a changefeed might be configured to wait for N changes before sending a response to the listening application.

Sharding and Replication

RethinkDB features a web-based administration dashboard that simplifies sharding and replication management shown in Figure 2.2, and offers information about data distribution as well as monitoring and data exploration features. RethinkDB uses B-Tree indexes and makes a special effort to keep them and the metadata of each table in memory, so as to improve query performance. Cache memory size in RethinkDB is set automatically in accordance with total memory size, but can also be manually configured.

RethinkDB implements sharding automatically, evenly distributing documents into shards using a range sharding algorithm [14] based on table statistics and parameterized on the table's primary key. In the event of primary replica failure, one of the secondaries is arbitrarily selected as primary, as long as a majority of the voting replicas for each shard remain available. To overcome a single-node failure a RethinkDB cluster must be configured with three or more servers, tables must be configured to use three or more replicas, and a majority of replicas should be available for each table. If the failed server is a backup, performance is not affected as progress is possible with a surviving majority.

An important management feature offered by RethinkDB is its programmatic cluster management API [25] accessible via ReQL queries to automate different types of reconfiguration. Internally, RethinkDB maintains a number of system tables that expose database settings and the internal state of the cluster. An administrator can use ReQL through a language like Javascript to query and interact with system tables using conventional ReQL commands, just as with any other RethinkDB table. To exercise granular control over sharding and replication, the administrator uses the *table_config* table as shown in Figure 2.1, which contains documents with information about the tables in a database cluster, including details on sharding and replication settings.

RethinkDB offers three primary commands for managing shards and replicas:

- *table_create*, for creating a table with a specified number of shards and replicas.
- *reconfigure*, for changing sharding and replication settings of a table (number of shards, number of replicas, identity of primary), after determining the current status via *table_config*. The location of replicas can be changed via updates to *table_config*. An example shown in Listing 2.1 where a reconfiguration applied

to a table, creating four shards with three replicas each.

Listing 2.1: Table Reconfiguration

```
r.table('users').reconfigure(shards=4,
        replicas=3).run(conn)
```

- *rebalance*, for achieving a balanced assignment of documents to shards, after measuring the distribution of primary keys within a table and picking split points. Listing 2.2 illustrates an example of this case.

Listing 2.2: Table rebalance

```
r.table('users').rebalance()
```

Consistency guarantees

RethinkDB chooses strong defaults for update consistency, and weak defaults for reads. By default, updates (inserts, writes, modifications, deletes, etc) to a key are linearizable, which means they appear to take place atomically at some point in time between the client's request and the server's acknowledgement. For reads, the default behavior is to allow any primary to service a request using its in-memory state, which could allow stale or dirty reads.

RethinkDB defaults will not acknowledge writes until they're fsynced to disk. Users may obtain better performance at the cost of crash safety by relaxing the table's or request's durability setting from hard to soft. As with all databases, the file system, operating system, device drivers, and hardware must cooperate for fsync to provide crash safety. In this analysis, we use hard durability and do not explore crash safety.

RethinkDB allows tuning write safety at *table* level by using the *table_config* table and setting *write_acks* to one of the following values:

- *majority*, is the default option meaning writes will be acknowledged when a majority of replicas have confirmed their writes.
- *single*, meaning writes will be acknowledged when a single replica acknowledges it

Unlike write safety, read safety is controllable on a *per-request basis* and there are three available options:

- *single* is the default and returns values that are in memory (but not necessarily written to disk) on the primary replica
- *majority* will only return values that are safely committed on disk on a majority of replicas. This requires sending a message to every replica on each read, so it is the slowest but most consistent.
- *outdated* will return values that are in memory on an arbitrarily-selected replica. This is the fastest but least consistent.

Finally, RethinkDB lets the administrator control the durability of a table. This can be done at *table* level by utilizing the *table_config* and setting *durability* mode option to one of the following values:

- *hard* writes are committed to disk before acknowledgements are sent
- *soft* writes are acknowledged immediately after being stored in memory

Other Features

RethinkDB provides some other features as well which listed below.

- *server tags* used to associate table replicas with specific servers, physical machines, or data centers.
- *dump* command line utility allows the data export from a live cluster

2.2.2 Containers and Docker

Containers

A *container* image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.

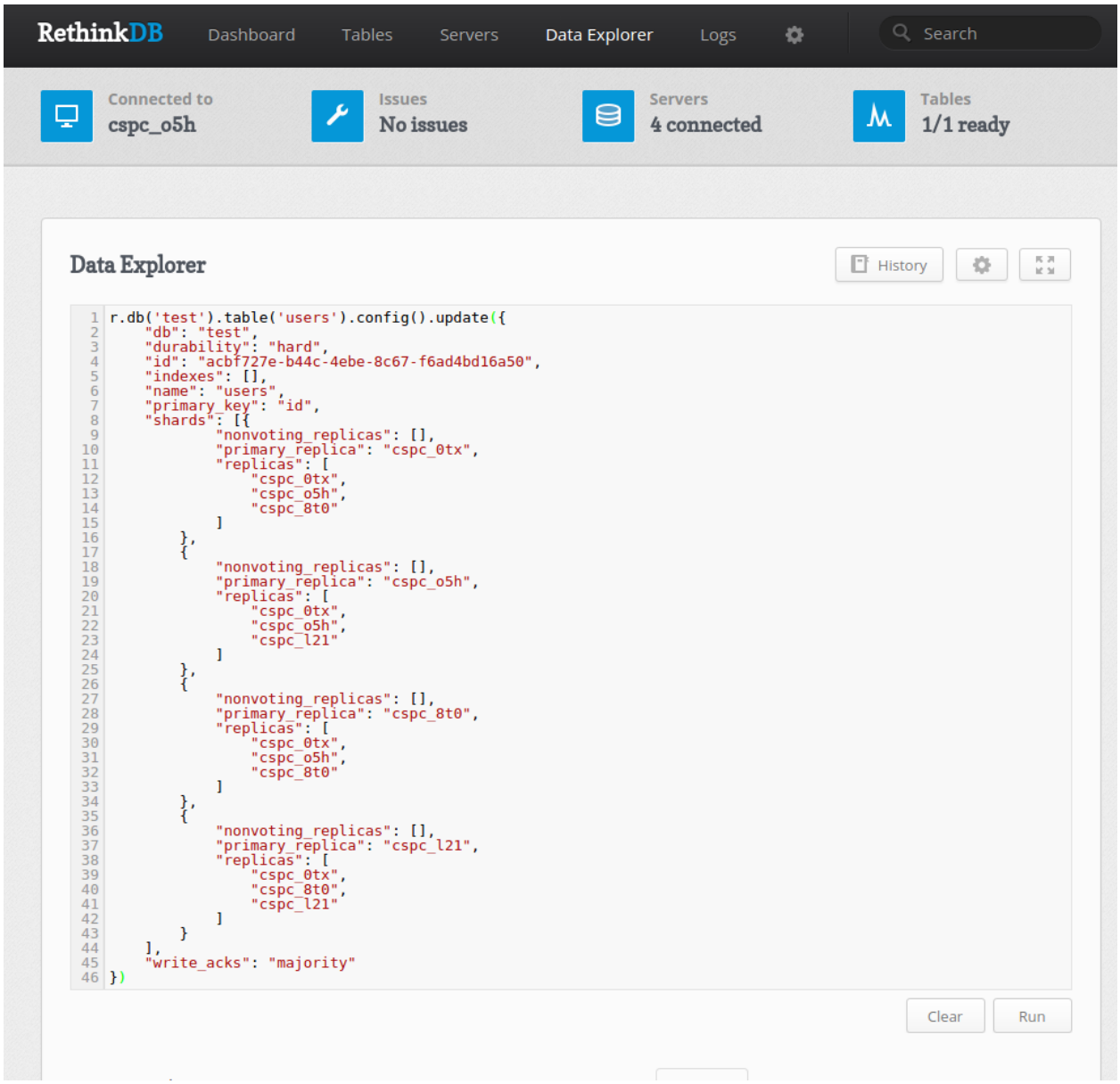


Figure 2.1: Updating the contents of *table_config* via the web console

Docker

Docker [39] is a tool designed to make it easier to create, deploy, and run applications by using containers. Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application. Docker Engine on Linux relies on another technology called control groups [40] (cgroups). A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints.

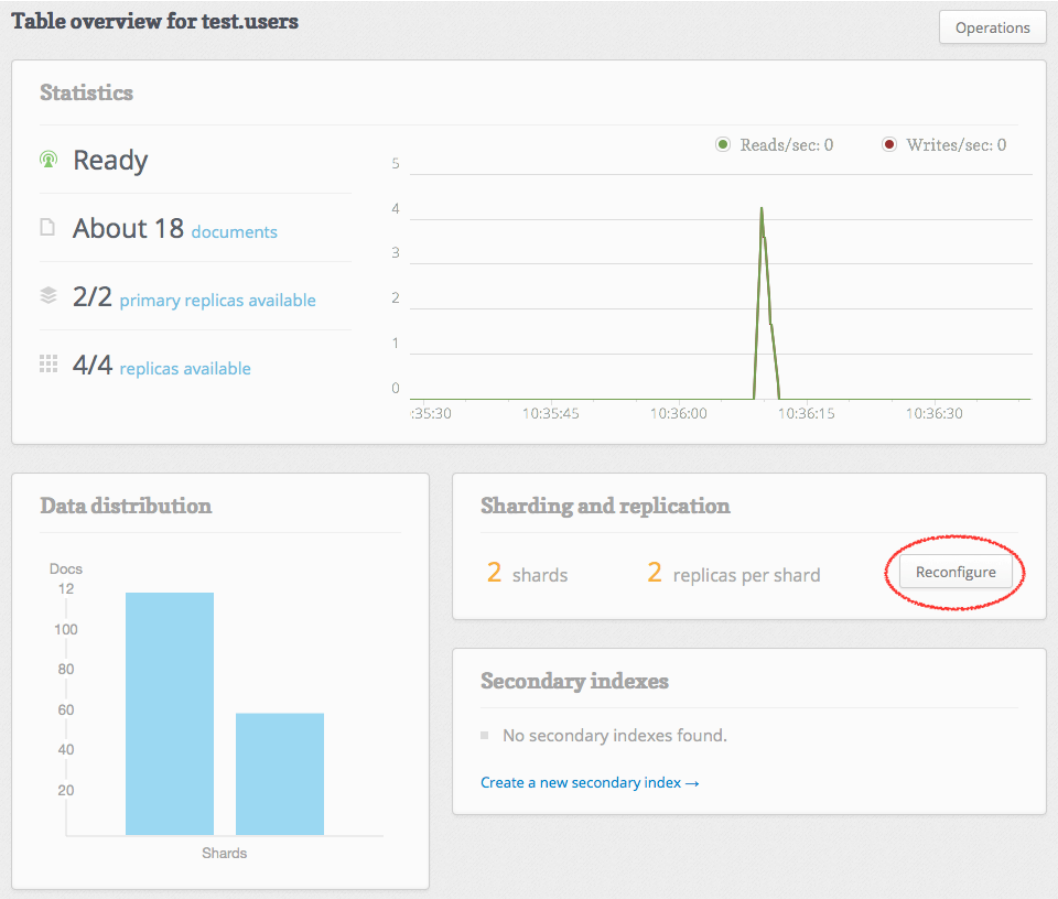


Figure 2.2: Sharding and replication via the web console [1]

2.2.3 Kubernetes

Kubernetes [19] is an open-source platform for automating deployment, scaling, and operations of application containers (such as Docker) across clusters of hosts, providing container-centric infrastructure. It supports more container runtimes than just Docker, however Docker is the most commonly known runtime, and it helps to describe pods in Docker terms. Kubernetes architecture is depicted in Figure 2.3 [2]. At its core is a shared persistent store (*etcd* [41] in Figure 2.3) with components watching for changes to relevant objects. State in Kubernetes is accessed exclusively through a domain-specific REST API that applies higher-level versioning, validation, semantics, and policy, in support of clients.

In Kubernetes, servers that perform work are known as nodes. Node servers have a few requirements that are necessary to communicate with the master components, configure the networking for containers, and run the actual workloads assigned to them. Each Kubernetes node comprises an on-machine agent called a *kubelet*, and

a component analyzing resource usage and performance characteristics of running containers called *cadvisor*. Kubernetes core components are the following:

Pods

A *pod* is the basic building block of Kubernetes, the smallest and simplest unit in the Kubernetes object model that can be created or deployed. Represents a running process on the cluster. It encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the containers should run. A Pod represents a unit of deployment that can be a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.

Service

A *service* is a unit that acts as a basic load balancer and ambassador for other containers. A service groups together logical collections of pods that perform the same function to present them as a single entity. It is an interface to a group of containers so that consumers do not have to worry about anything beyond a single access location. By deploying a service, administrator can be gain discover-ability and can simplify his container designs.

Replication Controller

A more complex version of a pod is a replicated pod. These are handled by a type of work unit known as a *replication controller*. A replication controller is a framework for defining pods that are meant to be horizontally scaled. The replication controller is delegated responsibility over maintaining a desired number of copies. This means that if a container temporarily goes down, the replication controller might start up another container. If the first container comes back online, the controller will kill off one of the containers.

ReplicaSet

ReplicaSet is the next-generation Replication Controller. The only difference between a ReplicaSet and a Replication Controller is the selector support. ReplicaSet supports a

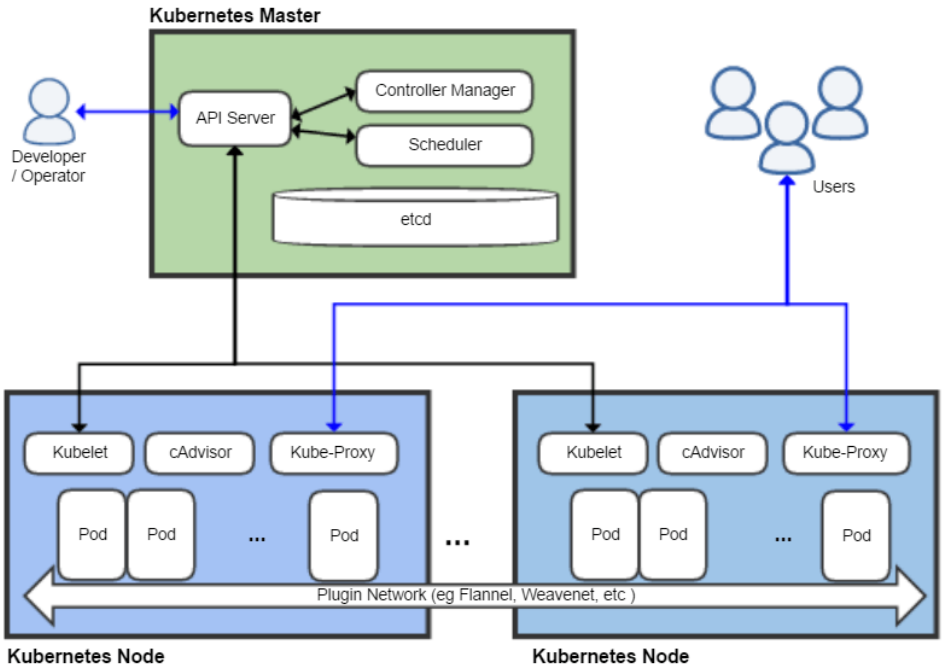


Figure 2.3: Kubernetes architecture [2]

newer set-based selector requirements whereas a Replication Controller only supports equality-based selector requirements.

Deployments

Deployments are intended to replace Replication Controllers. They provide the same replication functions (through Replica Sets) and also the ability to rollout changes and roll them back if necessary.

Horizontal Pod Autoscaler

Horizontal Pod Autoscaliner [42] was built to automate elasticity actions. Using an autoscaling algorithm automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization (or, with alpha support, on some other, application-provided metrics). The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user. The Horizontal Pod Autoscaler is not utilized in our case since is not applicaple for isolated performance hot-spots

Chapter 3

Design

3.1 Architecture

Our goal in this thesis is to explore the benefits possible through the use of *replica groups reconfigurations* in achieving service-level objectives over NoSQL systems. Responding to overload conditions in data stores has been addressed in previous work using control theory, by means of elasticity, rebalancing, and throughput arbitration controllers [8, 43, 44]. Lim *et al.* [8] showed that it is possible to use an integral controller for increasing or decreasing cluster size in response to average server CPU exceeding or dropping under a target (reference) threshold. An associated rebalancing controller regulates data movement bandwidth aiming to achieve time / performance impact goals. Karlsson *et al.* [43] showed that it is possible to use an adaptive integral controller to arbitrate system throughput via throttling actions, achieving service differentiation, when the overall system is operating close to capacity. However, while throttling is an effective short-term remedy and a way to enforce performance differentiation, an underprovisioned system should in the long term be expanded via an elasticity action.

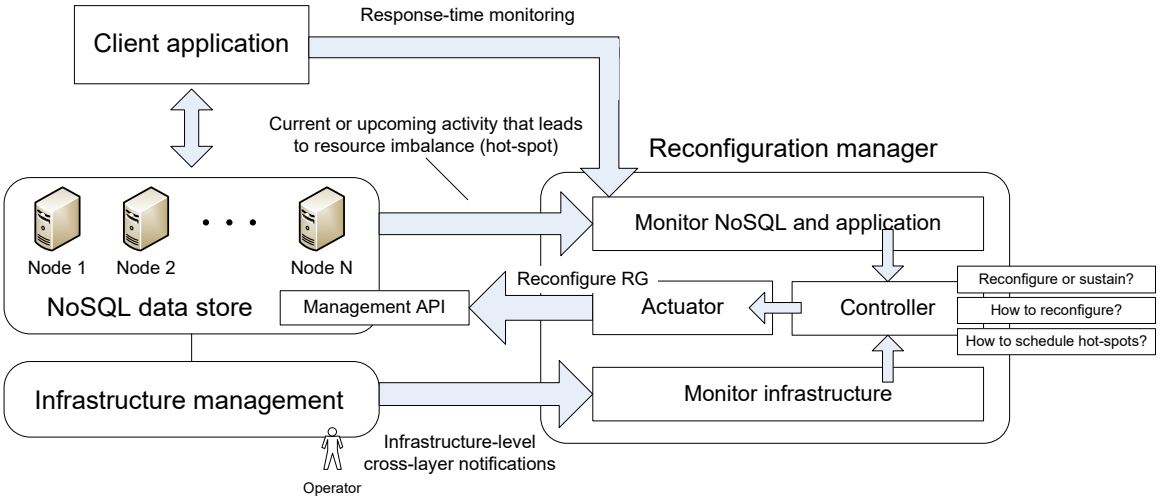


Figure 3.1: Reconfiguration manager architecture

In this thesis we consider a case of performance overload where neither an elasticity nor a throttling action offers an effective remedy. This is the case of a temporary *performance hot-spot*, either due to an external cause that reduces node capacity or due to a periodic background activity siphoning off node resources, where paying the cost of rebalancing (either moving data to a new node or rebalancing among the existing nodes) is not an effective solution, as the hot-spot is not expected to last long enough relative to the time-scale needed to rebalance data. This is a case where *replica-group reconfigurations* are expected to be an appropriate action to take to regulate a system towards maintaining a target (objective) response time for client workloads.

Figure 3.1 depicts the architecture of the *reconfiguration manager*. Monitoring is performed at the application, NoSQL data store, and infrastructure levels. While the manager may have monitoring access to the application and NoSQL data store layers through well-defined APIs, communication with the infrastructure management system is typically harder and not always available. Section 5.3 exemplifies such communication via notifications sent by a container management system to the reconfiguration manager, however the reconfiguration manager may operate even without such communication.

The manager monitors response-time at the application layer and CPU usage across nodes at the NoSQL data store. If response time violations are detected in conjunction with skewed overload conditions of a temporary nature (duration T) on a node, the controller may decide to actuate replica-group reconfigurations moving primaries off of that node. Note that the manager cannot rely on response-time mea-

surements alone (as it may be unable to determine whether replica-group reconfigurations are the appropriate type of response to exceeding a response-time objective), or CPU measurements alone (as an isolated overload condition may not necessarily lead to violation of a response-time objective).

We define as skewed overload the condition where some node N_i experiences high utilization (U) (exceeding 80%) while also exceeding the cluster average utilization by 30%:

$U(N_i) \geq 80\%$ and
$U(N_i) \geq 130\% \times \text{average}(U(N_1), \ldots, U(N_n))$

The constants 80% and 130% have been empirically determined to work well in detecting node overload conditions, distinguishing from a uniformly overloaded system that would justify increasing capacity through an elasticity action. Determining the exact values that would maximize the benefit from adaptation actions is a subject of ongoing research.

The time and placement of hot-spots can be known in advance when they are expected to happen (or scheduled) at specific times. Their duration can often be estimated via straightforward calculations, for example when nodes perform data-related reorganization and management activities such as compactions [15] and their duration depends on the amount of data processed, available network bandwidth, etc. Workload prediction has been demonstrated in the past using ARIMA-based time-series prediction [45].

The controller must decide when to apply reconfiguration(s) and to ensure that their benefit outweighs their cost. In general, the key action points for the reconfiguration controller are:

1. When to reconfigure shard(s): It is cost-effective to reconfigure when the primary is heavily impacted for some amount of time and the benefit of reconfiguration outweighs the cost (mainly the outage incurred due to leader elections). The cross-over point between reconfiguring or sustaining the hot-spot depends on the intensity of the hot-spot as well as on the load level of the NoSQL data store. In general it is possible to adaptively determine the cross-over point through online measurements as demonstrated in our experimental evaluation.

2. Where to move each shard's primary: Primaries should be moved to machines not experiencing a hot-spot, in a balanced manner. Note that is not always possible

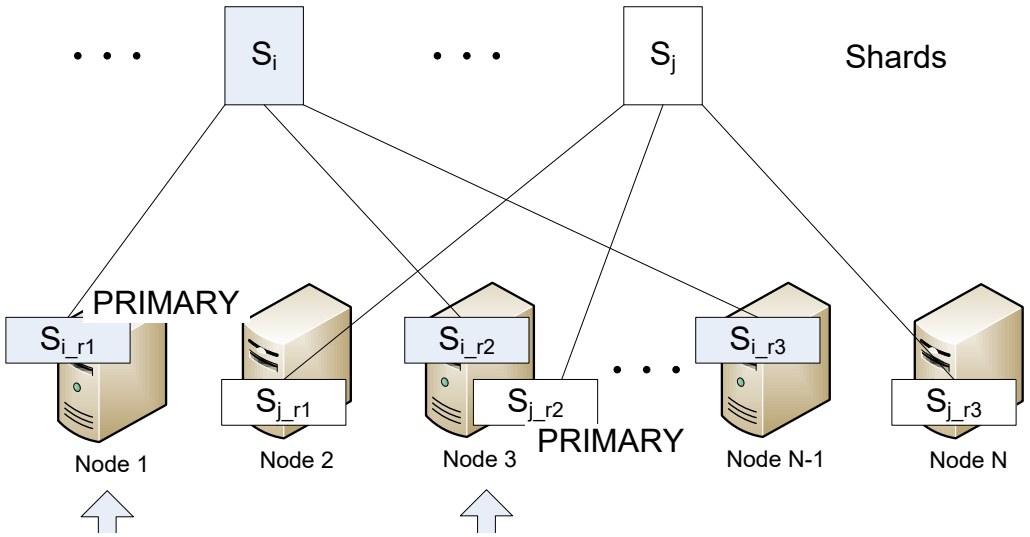


Figure 3.2: Two hot-spots apply simultaneously on nodes 1 and 3

to spread primaries evenly across all normally-loaded nodes, since available options are constrained by the current locations of the replicas.

3. How to schedule hot-spots, if such control is possible: When such control is possible, the aim should be to avoid simultaneously impacting a majority of replicas of any shards.

In Section 5.3 we will exhibit an experimental methodology to determine the cost-benefit cross-over point for specific cases of workloads and data-store configurations and demonstrate the importance of points (2) and (3) in implementing an effective reconfiguration manager. To quantitatively compare the impact of a hot-spot to that of reconfigurations applied to mask the hot-spot, on a response-time service-level objective (SLO), we introduce a *penalty function (P)* that measures the effect of one or more actions on a SLO L, over a specific time interval T divided into n samples.

$$P = \sum_{i=1}^{n} \begin{cases} l_i - L, & \text{if } l_i > L \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

where l_i is the mean latency at the i-th sample, L is the latency target, and n is the number of samples over the selected time interval T. Intuitively, P measures the extent to which latency SLOs were violated by successful operations during T.

Figure 3.2 provides an example of the more intricate issues facing the manager. Two hot-spots H1 and H2 are impacting nodes 1 and 3 hosting the primaries of shards i and j (S_{i_r2} and S_{j_r2}) respectively. A reconfiguration controller could decide

to reconfigure shards i and j and move primaries away from nodes 1 and 3. Selecting any node currently not experiencing a hot-spot, or not about to experience one as far out in the future as possible, would be a good choice. Reconfigurations are expected to help when their benefit outweighs their cost (see evaluation Section 5.3), and when a majority of replicas are not simultaneously experiencing a hot-spot, since data-store operations typically require acknowledgments from a majority of nodes to make progress. Note that in Figure 3.2, shard S_i has two out of its three replicas on nodes 1 and 3, both impacted by hot-spots. Thus reconfiguration is not going to help with access to shard S_i. A solution to this problem is to delay H2 so that it does not overlap with H1, if such control (the ability to schedule the execution of hot-spots) is possible.

3.2 The impact of simultaneous hot-spots across data-store cluster

In this section we study the problem of simultaneous hot-spots from a theoretical standpoint and we aim to show that the potential of adaptation actions to mask load imbalances diminishes with an increasing number of simultaneous hot-spots. Thus, given values of N, for the number of nodes, S, for the number of shards, R for the number of replicas per shard, and H for the number of simultaneous hot-spots, we want to calculate the probability P_{total} that the hot spots affects more than majority(R) replicas of some shards is drastically increased as H increases. We start by calculating the probability $P(S_i)$ that the *i-th* shard S with R replicas is affected by the hot-spots. Considering that every hot-spot affects at most one node, $P(S_i)$ can be calculated with equations 3.2 and 3.3. Finally, using the addition rule for probability 3.5 we calculate P_{total}.

$$\binom{N}{H} = {}^{N}P_{H} = \frac{N!}{H!(N-H)!} \tag{3.2}$$

$$\binom{R}{R_{majority}} = {}^{R}P_{R_{majority}} = \frac{R!}{R_{majority}!(R-R_{majority})!} \tag{3.3}$$

$$P(S_i) = \frac{\sum_{i=R_{majority}}^{R} \binom{R}{R_{majority}} \binom{N-R}{H-i}}{\binom{N}{H}}, \quad H \geq R_{majority}\,. \tag{3.4}$$

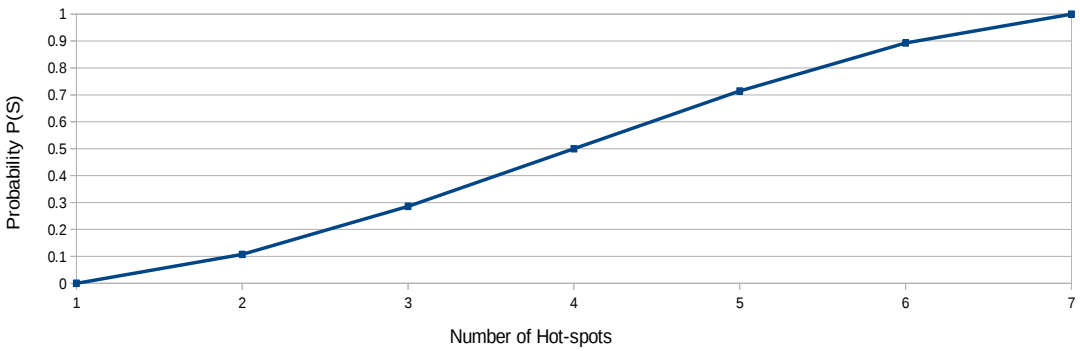


Figure 3.3: Probability of 8 different hot-spots to affect a majority of replicas of a particular shard consisting of 3 replicas on a 8 node cluster

$$P_{total} = P(S_1 \cup ... \cup \ S_s) = \sum_{i=1}^{S} P(S_i) - \sum_{\substack{i<j \\ i,j \in S}}^{S} P(S_i \cap S_j) + P(S_1 \cap ... \cap S_s) \tag{3.5}$$

Figure 3.3 depicts the probabilities that a number of simultaneous hot-spots (1 through 7) affects a majority of replicas of a particular shard consisting of three replicas on a eight node cluster. Using Equation 3.5 we can calculate the probability P_{total}, of at least a majority of any shard is affected by a hot-spot. In our evaluation we have experimentally determined that even with two randomly-placed simultaneous hot-spots on an 8-node cluster with 3 replicas per shard, replica-group reconfigurations cannot fully mask the effect of the hot-spots in 70% the cases (Section 5.3.2).

3.3 Handling infrastructure-related cross-layer notifications

The *reconfiguration manager* is able to intermediate between the infrastructure management and data-store layers as seen in the lower part of Figure 3.1. Its main purpose is to set goals regarding data store deployment (e.g., number of servers backing a cluster) and to reconfigure a cluster when notified about events of interest by the infrastructure layer, namely

- New server (pod) joined the cluster
- Server about to be decommissioned
- Server about to experience performance interference

Note that these are events that the NoSQL data store could not determine on its own without cross-layer notifications from the infrastructure management layer. They are typically produced by human operators or automatically by infrastructure management systems, such as the Kubernetes CMS used in this work. The next chapter provides implementation details.

Chapter 4

Implementation

4.1 Cross layer management

A full implementation of our cross-layer management architecture of Figure 3.1 comprises a container management system (CMS) at the lower layer, managing the infrastructure, and a containerized NoSQL data store at the top, deployed over a number of pods (Figure 4.1). For concreteness and without loss of generality, in what follows we describe our implementation using Kubernetes and RethinkDB concepts. However, the architecture can accommodate other CMS and data stores, such as the Amazon EC2 Container Service (ECS) and MongoDB respectively.

The resources (pods) of a data store cluster are described in a Kubernetes *deployment* spec. The manager uses Kubernetes' *ReplicaSet* mechanism [46] to set the desired goal on the number of pods that should be available to support the RethinkDB cluster. Kubernetes performs liveness monitoring through its internal heartbeat mechanism as well as application-specific *liveness probes* [47]. After detecting a pod failure, it deploys a new RethinkDB pod to maintain the replication[1] goal. The manager is implemented as a containerized node.js RESTful Web service, deployed in a dedicated

[1]In this context, replication refers to the number of RethinkDB servers backing a cluster, not the number of data replicas backing a shard.

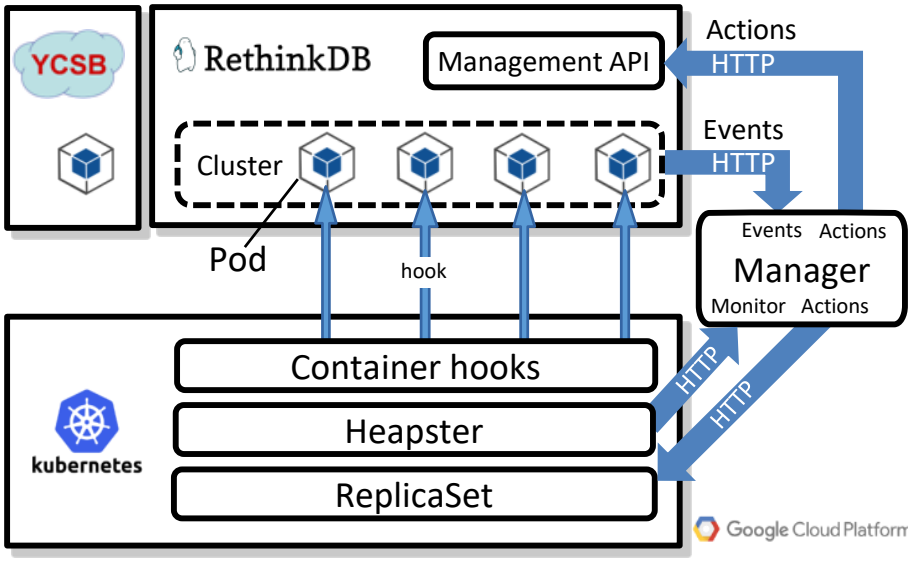


Figure 4.1: Reconfiguration manager implementation on Google Container Engine

pod managed by Kubernetes. The specific events it handles and actions it undertakes are described below:

Events. The manager exposes endpoints invoked by *container hooks* to communicate events relating to the infrastructure (pods) where RethinkDB is deployed. A hook provides information to a container about events in the container's management lifecycle. As soon as a hook is received by a container's hook handler, a HTTP call is executed to the corresponding manager endpoint. We utilize two existing types of hooks:

- PreStop, raised before termination of a container. The container will not terminate until the manager acknowledges the HTTP call.
- PostStart, raised after the creation of a container.

We envision a third type of hook:

- PerfWarning, carrying notification of impending performance interference that will impact a specific pod.

The PerfWarning hook will enable a manager to respond with proactive adaptation actions aiming to mask the impact of such interference. To evaluate the benefits

of a PerfWarning hook, we approximate its functionality at the manager by periodically polling the *Heapster* container cluster monitoring service (Figure 2) to detect when a pod P_i in the cluster becomes resource-limited. Heapster, natively supported in Kubernetes, collects cluster compute resource usage metrics and exports them via REST endpoints. By polling Heapster periodically, the manager can detect skewed overload conditions (as defined earlier), namely a pod P_i exhibiting high utilization (exceeding 80%) while also exceeding the cluster average utilization by 30%:

$U(P_i) \geq 80\%$ and
$U(P_i) \geq 130\% \times \text{average}(U(P_1), \ldots, U(P_n))$

Actions. A PreStop hook is handled by the manager by demoting[2] all replicas on that pod. This will mask the impact of the pod's impending decommission: With success of read/write operations on RethinkDB (as well as many other replicated data stores) being dependent on the availability of a majority of replicas, loss of a single backup replica per shard (data partition) will not affect performance. Our evaluation (Sections 5.2.1 and 5.2.2) shows that the impact of reconfiguration (brief unavailability of a shard) is lower than that of unannounced node crash, as the latter includes the additional overhead (timeouts) of detecting the failure.

A PostStart may convey different types of events: First, it may be that the new pod is a replacement of a recent crash (i.e., restores the size of the cluster). The master needs to be involved here because although a newly bootstrapping RethinkDB server can check if the specific RethinkDB cluster it intends to join is online and then join it, the RethinkDB runtime cannot automatically reconfigure the cluster to utilize the new node. Thus in the event of a PostStart hook from a replacement node, the manager balances the cluster by migrating replicas to the new pod, restoring pre-crash state. If the new pod is increasing cluster size, which happens when expanding capacity during an elasticity action, the manager will decide to perform replica migrations to the new pod so as to increase the overall capacity of the cluster.

In the event that the manager detects an overload condition on one of the pods using monitoring (approximating a PerfWarning event) in an otherwise normally-loaded cluster, it decides to perform reconfigurations demoting primaries hosted on the overloaded node. As demonstrated in our evaluation (Sections 5.2.3 and 5.2.4), this provides rapid response to the performance degradation experienced by the

[2]Switch their role from primary to backup, electing other primaries.

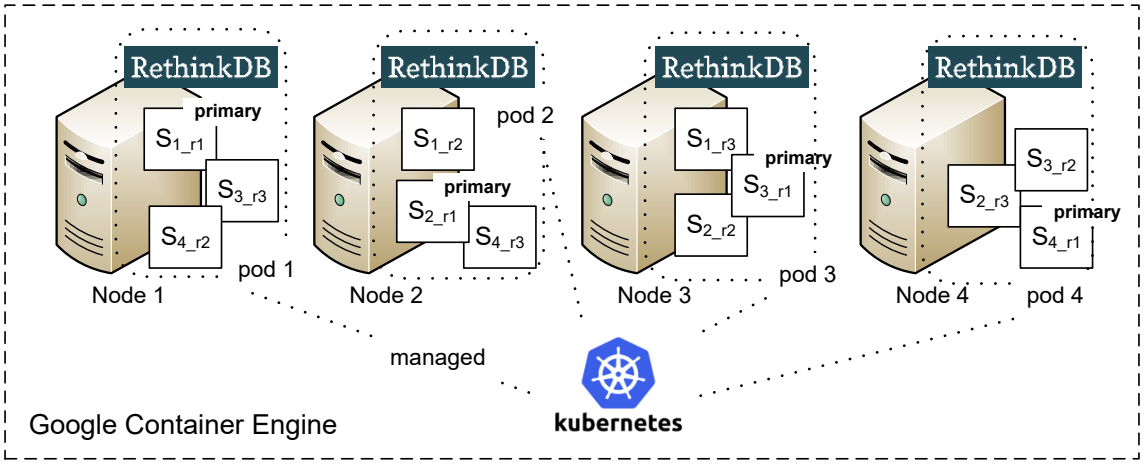


Figure 4.2: The experimental testbed

application.

The use of reconfiguration actions to rapidly ease an overload condition on a single node does have a drawback if the overload condition is expected to last a long time. Since replicas on the overloaded node are not updated frequently, a subsequent failure of a second node may leave some of the shards with less than a majority of live *and* up-to-date replicas, reducing their performance. This eventuality is demonstrated in the experiment of Subsection 5.2.5. Thus, if the manager receives feedback that an isolated overload condition is expected to last a long time, it should opt to reconfigure *and* migrate replicas outside of an overloaded node. Implementation of such functionality would require the ability to predict the duration of performance interference and convey it as a parameter to PerfWarning.

The manager invokes the RethinkDB management API to perform the following operations:

- *movePrimariesFromServer*: Demote primaries of all shards hosted at a given server. This is performed by retrieving the table configuration through *table_config* and then re-assigning shard primaries to other servers in a balanced manner by applying updates to *table_config*.

- *movePrimariesFromNode*: Demote primaries of all shards hosted at all servers of a given node. This is performed using the Kubernetes API to retrieve the servers (pods) placed at a given node and then applying *movePrimariesFromServer* on each of them.

- *addServer*: Include a new server to the configuration of a table using the *reconfigure*

command.

- *createReplica*: Create replica of a shard on a specific server (pod) updating the *table_config*.

The above described management architecture supports the following scenarios evaluated in this thesis:

Unscheduled downtime of a node. RethinkDB is able to adapt to a crashed server (pod) without the engagement of the manager through standard replication recovery mechanisms. When a new pod is made available by Kubernetes, the manager is notified through the PostStart hook, understands that it is replacement server, and proceeds to reconfigure the cluster.

Scheduled downtime of a node. In this case, the manager is notified when a pod is about to be decommissioned through the PreStop hook. It then reconfigures RethinkDB to demote all primaries hosted at that pod. When a new pod becomes available, actions are similar to the previous case.

Hot-spot node(s) impacting application performance. Through the Heapster monitoring API, the manager is able to detect a hot-spot on a node. In such case it demotes primaries hosted there. If the manager detects a uniformly overloaded status across the cluster, it decides to allocate a new pod and migrate a subset of replicas to it.

Our cross-layer management architecture was implemented as a service written in *Javascript* using *NodeJS* framework, about 600 lines of code (LOC). The service was containerized using Docker and deployed to cluster managed by Kubernetes CMS.

4.2 Reconfiguration controller

The reconfiguration manager is continuously monitoring for violations of the response-time SLO and for skewed overload conditions (hot-spots) on data-store nodes. It may also receive advance notification of hot-spots to occur in the data-store. When called to decide on whether to apply a reconfiguration or to sustain a hot-spot, the reconfig-

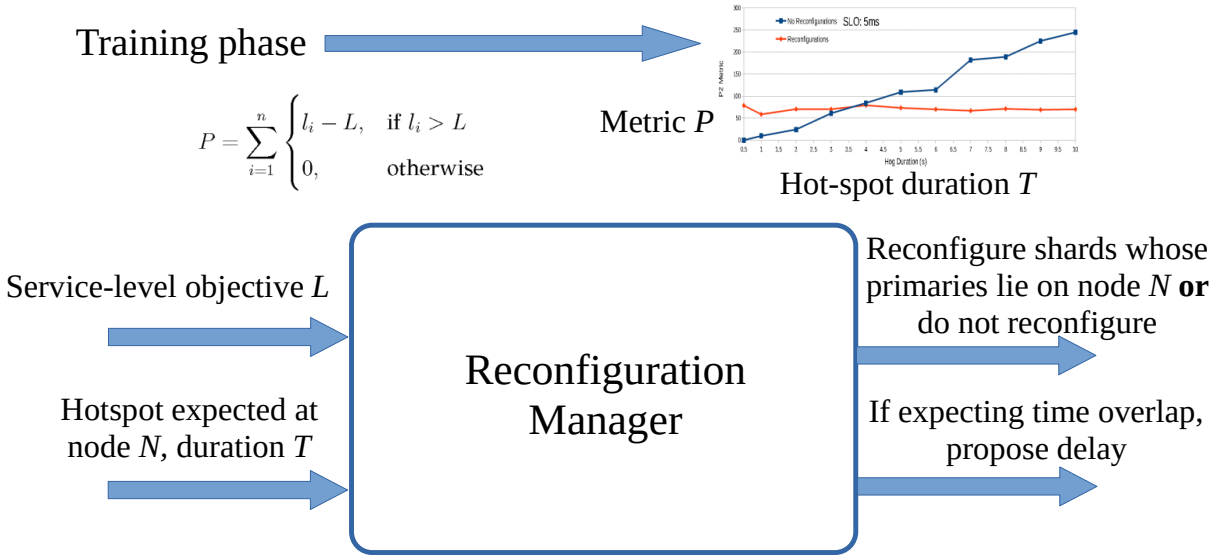


Figure 4.3: Reconfiguration manager architecture.

uration controller uses prior knowledge comparing the SLO penalty (Equation (3.1)) of that type of hot-spot compared to the SLO penalty of reconfigurations to mask that hot-spot (as for example in Figures 5.15 and 5.16). Key parameters are the duration and resource-intensity of the hot-spot and the fraction of data it affects.

Hot-spot intensity is always considered high in our experiments, consuming at least 90% of CPU. The duration of the hot-spot is an independent variable in our experiments as seen in Figures 5.15 and 5.16. The larger the fraction of the total data-set on shards whose primary lies on the impacted node, the stronger the impact of the hot-spot (and that of reconfigurations). The RethinkDB data store used in our current implementation initially creates as many shards as nodes (servers) in the cluster, thus the likelihood is that each node will be hosting the primary of a single shard at any time. Over time however, changes such as the additional or removal of nodes and replica-group reconfigurations, change the number of shard primaries per node. The size of each shard also varies over time. With multiple shards per node, the controller may decide to fine tune the amount of shards reconfigured, reducing the duration of outage (affecting a smaller fraction of the overall data) but also reducing the benefit as some shards will still have primaries on the overloaded node.

Figure 4.3 depicts the key elements of the reconfiguration manager. The manager receives as input a user-specified service-level objective (SLO) setting an upper bound

L on response time. Given this information, the reconfiguration manager starts collecting targeted systematic measurements, calculating the penalty function P for different hot-spot durations and different responses to the hot-spot, namely either trying to mask it with appropriate reconfiguration actions or not (sustaining the hot-spot). This outcome of this training phase is a graph such as depicted in the upper right corner of Figure 4.3 and studied experimentally in Section 5.3. Such graphs typically comprise two curves: One representing the increasing performance cost of sustaining increasingly long hot-spots, and another representing the near-constant cost of reconfiguring around the hot-spot. The cross-over point between the two curves determines the hot-spot duration after which the benefit of reconfiguration justifies its cost, and thus the controller will recommend applying it. While it may appear that computing this graph may be a time-consuming and performance-impacting process, our experimental evaluation indicates that cross-over points are typically found in short hot-spot durations (1-3 sec), thus in reality we expect to require measurements at 3-5 points to be able to pin-point the cross-over point. The possibility of pre-computing and storing such graphs is also practically relevant, however their computation over a training phase has the benefit of being adaptive to different system sizes, degrees of replication, SLOs, etc.

Assuming the controller has computed the penalty function vs. hot-spot duration graphs, it is able to decide on the appropriate action to take when receiving notifications about a forthcoming hot-spot to affect node N for a duration of T seconds. First, if the manager determines that the hot-spot may overlap with one or more other hot-spots about to take place, it may decide to recommend that it be postponed if possible. This decision is based on our theoretical analysis (Section 3.2) and experimental results (Section 5.3.2) suggesting that reconfiguration may not be able to fully mask several simultaneously occuring hot-spots. If no overlap is detected, the manager will use the penalty function vs. hot-spot duration graph to determine whether a reconfiguration on shards whose primaries lie on node N (where the hot-spot is going to act upon) is beneficial or not.

In our prototype we have implemented and experimented mainly with the training phase described above, involving setting up systematic measurements to compute penalty-function vs. hot-spot duration graphs and from them, determining the cross-over points. The evaluation of this part of our prototype is described in Section 5.3.

Chapter 5

Evaluation

5.1 Research questions

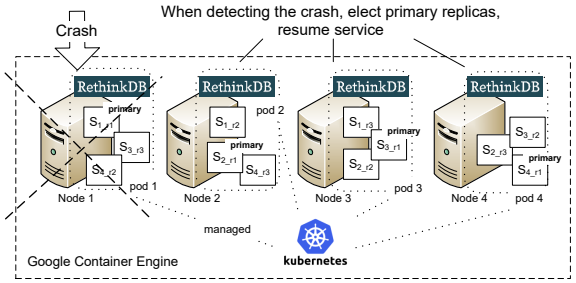
We perform a series of experiments aiming to address the following three research questions:

1. Does proactive adaptivity in the case of previously announced downtime of a node reduce the performance impact that would otherwise be experienced if the downtime was unscheduled (e.g., a crash)?

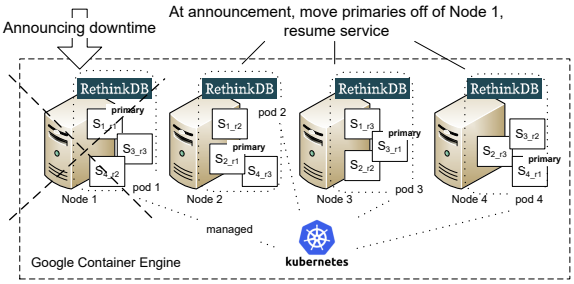
2. Can load balancing actions via replica group reconfigurations reduce the performance impact of *hot-spots* (overloaded nodes) on data store performance?

3. When does the cost of replica-group reconfigurations outweigh their benefits under different assumptions on hot-spot characteristics, especially their duration?

Research questions (1) and (2) are investigated in the context of the cross-layer management prototype deployed in the Google Container Engine (GCE) where Kubernetes

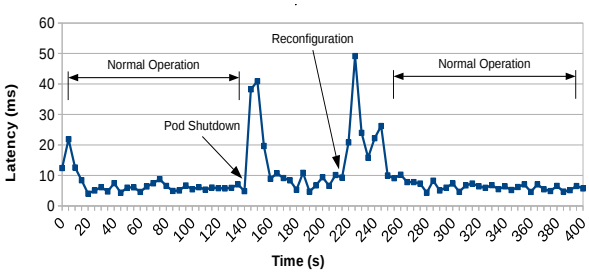


(a) Reactive adaptivity to unscheduled downtime of a node

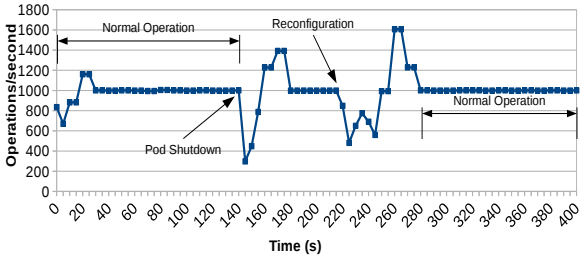


(b) Proactive adaptivity to scheduled downtime of a node

Figure 5.1: Reactive vs. proactive adaptivity to downtime of a node (Node 1)



(a) Mean operation latency during reconfiguration phase.



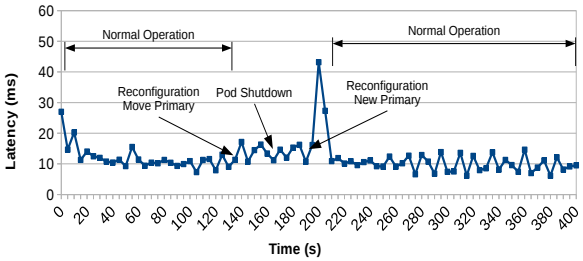
(b) Throughput during the reconfiguration and restoration phase.

Figure 5.2: Reactive adaptivity to unscheduled downtime of a node (scenario of Figure 5.1a)

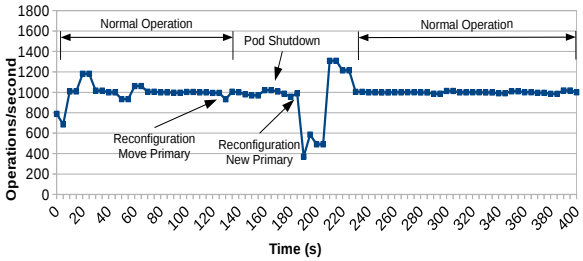
is natively supported, whereas question (3) is investigated in a dedicated private cluster to avoid resource constraints in GCE and to achieve the low statistical noise needed in these experiments.

5.2 Evaluation of the cross-layer management prototype

Experimental testbed. The experiments were conducted on a 4-node container cluster on Google Cloud Engine. Each node has 2 vCPUs (2.6 GHz Intel Xeon E5 with hyperthreading), 13GB of RAM, and a solid-state drive (SSD) for persistent storage. Each GCE node hosts one Kubernetes pod containing a RethinkDB server (Figure 4.2). RethinkDB is configured for 4 shards. Each pod initially contains 3 replicas of 3 different shards. Each replica is denoted S_{i_rj}, which stands for "shard i, replica j", where $i \in \{1,2,3,4\}$ and $j \in \{1,2,3\}$. The YCSB benchmark [48] is



(a) Mean operation latency during reconfiguration phase.



(b) Throughput during the reconfiguration and restoration phase.

Figure 5.3: Proactive adaptivity to scheduled downtime of a node (scenario of Figure 5.1b)

configured for Workload A (50% reads, 50% updates) [49], 16 client threads, and load target of 1000 operations/sec. RethinkDB is configured for soft durability (writes are acknowledged immediately). The software versions used are Kubernetes 1.4.8 and RethinkDB 2.3.5.

We first look into the benefits of proactive vs. reactive adaptation when a RethinkDB server is decommissioned. We note that proactive adaptation is only possible with a previous announcement of downtime, while reactive adaptation makes sense only in the case of unscheduled downtime (e.g., crash).

5.2.1 Reactive adaptivity to unscheduled downtime of a node

This scenario is illustrated in Figure 5.1a. Figure 5.2a depicts YCSB latency and throughput during execution. At 150s a server is decommissioned ("pod shutdown") causing all replicas hosted there to crash. Crashed primary replicas cause a brief period of unavailability, out of which RethinkDB recovers (at 165s) by electing a new primary for those crashed. At 180s a new node is made available by Kubernetes (via the ReplicaSet mechanism) and joins the RethinkDB cluster. The manager initiates the creation of replicas at the new node via a reconfiguration of the YCSB table, causing RethinkDB to start *backfilling* (transfering state to) simultaneously all the replicas. After the replicas are created, some of them are assigned primaries to restore balance. The performance impact of this process is seen at 220s–260s.

5.2.2 Proactive adaptivity to scheduled node downtime

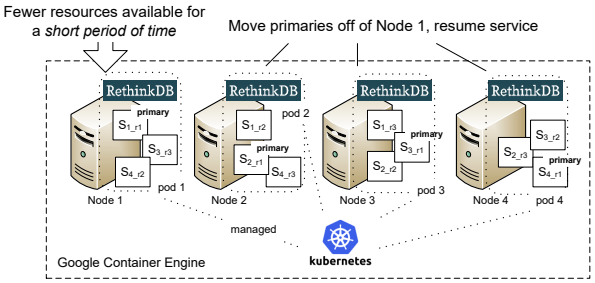
A different scenario involving a scheduled maintenance activity that will take down a node (Node 1) is illustrated in Figure 5.1b. Unlike the previous case, the manager is notified in advance and thus reconfigures the RethinkDB table to demote any primaries hosted at Node 1 (electing primaries at other nodes), aiming to reduce impact on cluster performance. In Figure 5.3a we observe that reconfiguration takes place at 140s. Decommissioning Node 1 at 170s has only a small impact on performance.

The new server joins the cluster at 200s, at which point the manager starts backfilling replicas there. Finally, the manager reconfigures the table again at 195s to promote an appropriate number of backup replicas into primaries (one in this case) on the new server to reach a well-balanced table configuration. Comparing Figure 5.2 to Figure 5.3 makes it clear that proactive adaptivity leads to smoother performance overall.

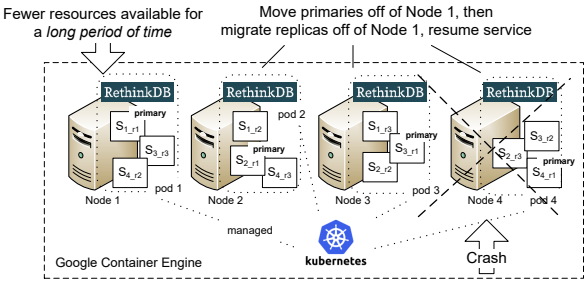
We next turn our attention to adaption actions performed by the manager in the context of a load imbalance on some node in the cluster. To evaluate the benefits of adaptation in this case, we set up a third experiment in which a resource-intensive process kicks in on one of the nodes in the cluster at some point in time, draining CPU resources and impacting the overall performance of the cluster. The cause of the impact is the fact that the affected node hosts the primary replica of one of the shards in the cluster, and thus all operations addressing this shard are being delayed. This challenge can be addressed as described below:

5.2.3 Offloading a brief hot-spot via reconfiguration

The next scenario is illustrated in Figure 5.4a. As shown in Figure 5.5a, at 100s a resource-intensive activity drains resources on a cluster node, with significant impact on overall cluster performance as operation latency surges from about 5ms to 80-120ms while throughput drops from 1000 ops/sec to 200 ops/sec. Drastically reduced performance is due to those shards (replica groups) whose primaries are hosted at the affected node. At about 170s the manager decides to start a reconfiguration of RethinkDB cluster nodes, demoting any primaries hosted at the affected node and electing primaries in other nodes in the cluster (making sure to spread load

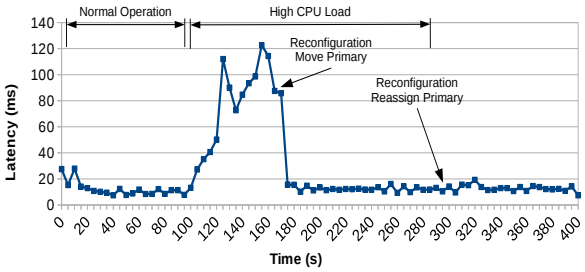


(a) Temporary offload via replica-group reconfiguration

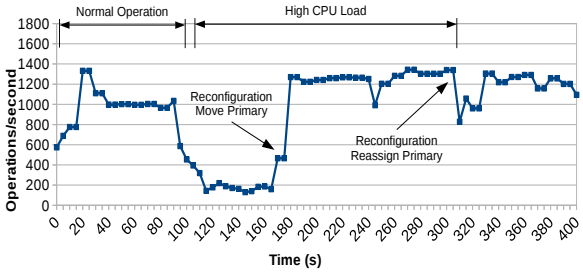


(b) Long-term offload: replica reconfiguration *and* migration

Figure 5.4: Offloading a hot-spot (Node 1): Brief vs. long-lasting hot-spot



(a) Mean operation latency during reconfiguration phase

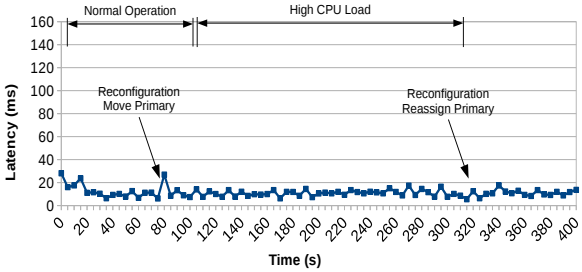


(b) Throughput during the reconfiguration and restoration phase

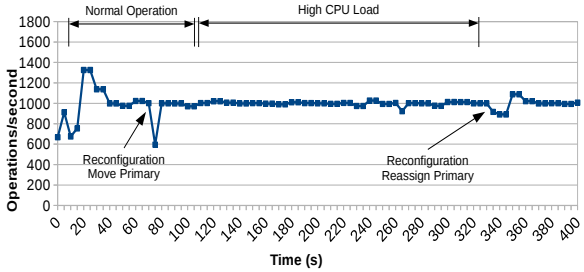
Figure 5.5: Offloading a brief hot-spot via reconfiguration (scenario of Figure 5.4a)

uniformly). The affected node now hosts only backup replicas, no longer posing an impact on performance: Each shard is backed by three replicas, and any two nodes (some majority) are sufficient for reads/writes to the shard to make progress.

After reconfiguration of the cluster, latency is restored to about 10ms, significantly better than during the surge (80-120ms). As the YCSB node is producing load at a constant rate of 1000 ops/sec, service-side request queues are building a backlog during the surge. The throughput rise at 190s is due to the emptying of those queues. At 310s, the manager initiates a reconfiguration of the cluster back to its original state, promoting a replica to a primary on Node 1, better balancing load. This brings latency down to pre-surge levels of about 10ms. CPU utilization at Node 1 during the reconfiguration and restoration phase is illustrated in Figure 5.8.

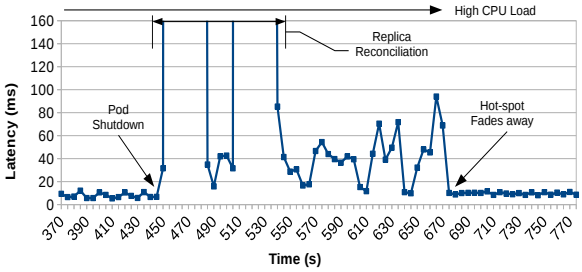


(a) Mean operation latency during reconfiguration phase

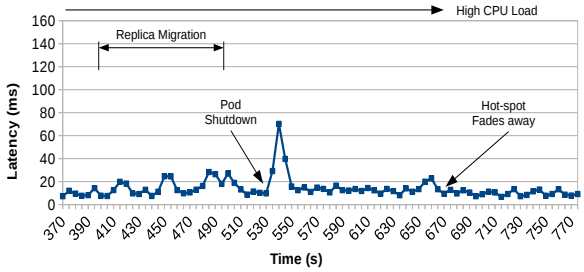


(b) Throughput during the reconfiguration and restoration phase

Figure 5.6: Offloading a brief hot-spot via proactive reconfiguration (scenario of Figure 5.4a)



(a) Mean operation latency during server failover without migrating backups



(b) Mean operation latency during server failover after migrating backups

Figure 5.7: Offloading a brief hot-spot via proactive reconfiguration (scenario of Figure 5.4a)

5.2.4 Offloading a brief hot-spot via proactive reconfiguration

This scenario shows the benefit of applying the reconfiguration action proactively, in advance of the hot-spot. As shown in Figure 5.6, a reconfiguration is applied at 80s prior to the resource-intensive activity coming into effect at that node at 110s. As our implementation does not yet support a predictive implementation of PerfWarning event (Section 4.1), this is triggered manually in this case. The proactive action is able to mask completely the adverse effect of the hot-spot on application performance. After the end of the resource-intensive activity, another reconfiguration restores the replica roles (promoting a primary) on the affected node.

Reconfiguration by reassigning shard primaries is an effective short-term remedy to maintain performance in the presence of a temporary hot-spot. However, a downside is that it leaves the system vulnerable to service unavailability in the case of a

subsequent crash, as is demonstrated next:

5.2.5 Offloading a longer-term hot-spot via reconfiguration and replica migration

The next scenario is illustrated in Figure 5.4b. This experiment focuses on the time following the movement of primaries off of Node 1 in the previous experiment and while the hot-spot is still on (lasts longer in this case). To focus on the behavior during a subsequent crash we show results starting at 370s into the run. We focus on the two possible outcomes that may occur when a second node (Node 4) crashes, depending on whether new instances of the replicas located in Node 1 have been created on other nodes prior to the crash on Node 4 (Figure 5.7b) or not (Figure 5.7a).

In Figure 5.7a we observe that the pod crash on Node 4 at about 450s results in severe performance degradation. The crash of Node 4 brings down 3 replicas (of different shards), one of which is a primary and two are backups. Shards with replicas in Nodes 1 and 4 are now left with two replicas, one of which outdated (on Node 1) since replicas there were not updated quickly enough. Reads are thus delayed in the interval 450s-550s while the slow replicas are "catching up" (being reconciled) through backfilling. Even after reconciliation (550s-680s), replicas at the hot-spot node continue to delay progress, since the shard depends on them to form a majority.

We next consider the case where the manager *migrates* replicas out of Node 1 (400s-500s in Figure 5.7b), an action that poses a slight latency cost during data transfer. The migration lasts for about 100s as the replicas are migrated one by one to reduce the performance impact. However, crash of a pod at 535s has now little effect on performance, as all shards have at least two up-to-date replicas to satisfy read and write operations from. Replica migrations are a worthwhile action in this case since the hot-spot lasted longer and the system was more vulnerable to a subsequent crash.

When possible, providing the manager with knowledge on the time duration of a hot-spot can guide the choice of an appropriate policy: reconfigure shards (demote primaries on a hot-spot node) if the hot-spot is expected to be short, otherwise reconfigure *and* migrate replicas out of a hot-spot node.

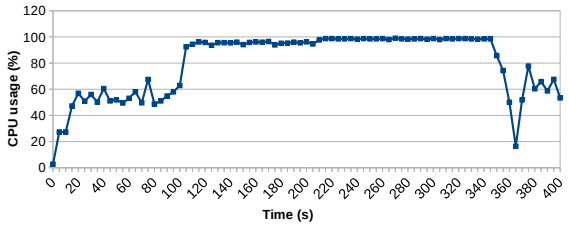


Figure 5.8: CPU utilization on hot-spot node (including all activities)

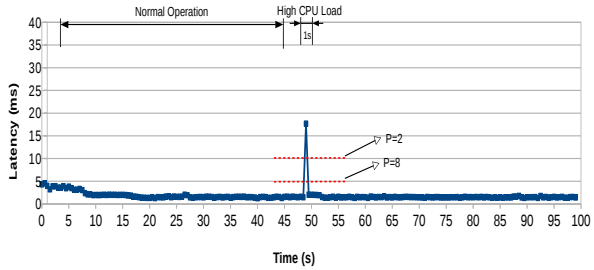
5.3 Cost-benefit analysis of adaptation actions

In previous experiments we considered hot-spots of a long duration, where the reconfiguration action always turned out to be beneficial. However, for shorter hot-spot spikes a question that comes up is whether the cost of the reconfiguration action itself (the availability lapse incurred by the leader switch) may outweigh the benefits. In what follows, we carry out an experimental cost-benefit analysis considering both simple hot-spot spikes as well as concurrent occurences of such spikes across cluster nodes.

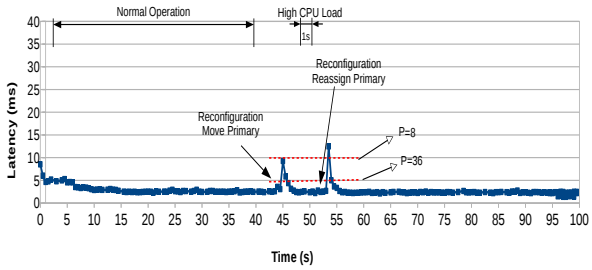
Experimental testbed. Our experimental testbed is a cluster of 9 servers, each equipped with a dual-core (four hardware threads) AMD Opteron™275 processor clocked at 2.2GHz with 12GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers used as RethinkDB data nodes store data on a dedicated 300GB 15,500 RPM SAS drive, so as to not interfere with the base 72GB 10,000 RPM SCSI drive used by the OS. All hard drives are formatted with ext4.

5.3.1 Determining the break-even point for individual hot-spot spikes

In this scenario we aim to determine the point at which the cost of a replica-group reconfiguration on performance outweighs its benefits in masking a temporary hot-spot. To achieve this we perform experiments with hot-spots of varying duration and compare their impact on performance (without a reconfiguration) to the impact of a reconfiguration action prior to the hot-spot with a subsequent reconfiuguration restoring the balance of replica roles on the cluster. To quantitatively compare the two

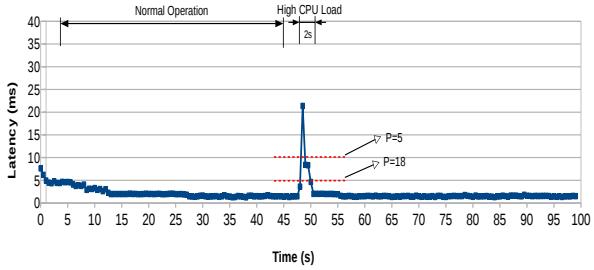


(a) Normal operation without reconfiguration

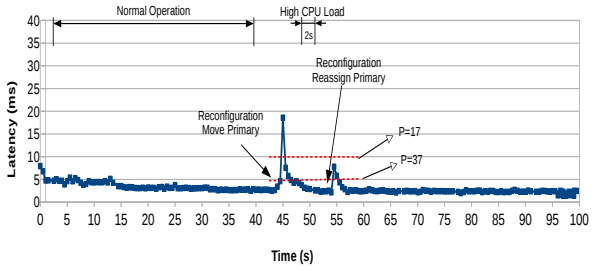


(b) Proactive reconfiguration and restoration

Figure 5.9: Mean operation latencies under a 1s hot spot (95% reads - 5% writes, Configuration C1))



(a) Normal operation without reconfiguration



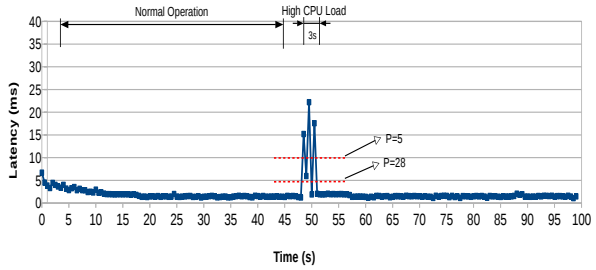
(b) Proactive reconfiguration and restoration

Figure 5.10: Mean operation latencies under a 2s hot spot (95% reads - 5% writes, Configuration C1))

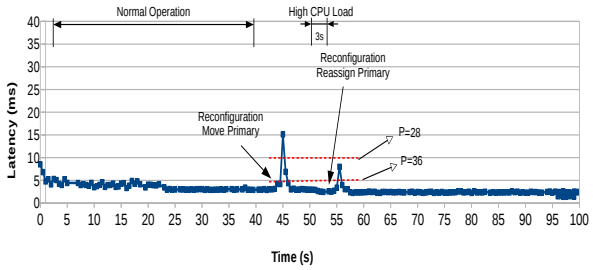
cases we use the *penalty function (P)* introduced in Equation (3.1) using a sampling period of 500ms.

We calculate P during hot-spots by selecting T such that it fully includes the duration of the hot-spot, divide T to 500ms samples, set l_i to the average measured response-time during sample i, and directly apply the penalty function. One issue we had to face was how to calculate P when reconfigurations were involved during T, as operations issued during reconfigurations fail. To achieve this, we measure the number of failed operations, say 1200 (or roughly 600 during each reconfiguration) on a run that averaged 6000 ops/sec, and thus deduce that each reconfiguration lasts about 100ms. One question we faced was what response-time to attribute to the 600 operations that failed, given the fact that RethinkDB throws an exception to them and those operation are counted as unsuccessful[1]. We decided to consider the 600

[1]YCSB does not reissue failed operations, it rather gives up on them and continues with the issuing



(a) Normal operation without reconfiguration



(b) Proactive reconfiguration and restoration

Figure 5.11: Mean operation latencies under a 3s hot spot (95% reads - 5% writes, Configuration C1))

Name	# Nodes	Read policy	Write policy	Durability	Replicas/shard	Data-set size (rows)
C1	8	primary replica	primary replica	soft	3	1 million
C2	8	primary replica	majority	soft	3	1 million
C3	4	primary replica	primary replica	soft	3	1 million
C4	8	primary replica	majority	soft	3	6 million

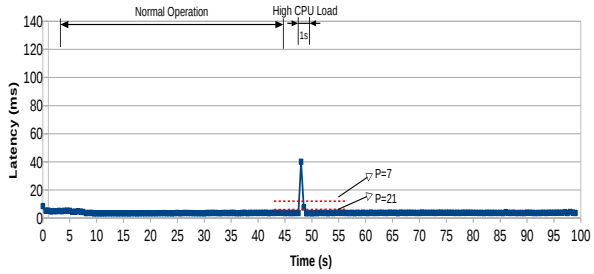
Table 5.1: Summary of RethinkDB configurations

operations that failed during a reconfiguration as a single operation that lasts for the full amount of the outage (i.e., 100ms). This is a reasonable assumption as an application that would persist on re-issuing the operation would have succeded to complete it at the end of the reconfiguration period. Since we use 500ms samples and reconfiguration takes 100ms of one such sample, we calculate the mean latency l_i of that sample as

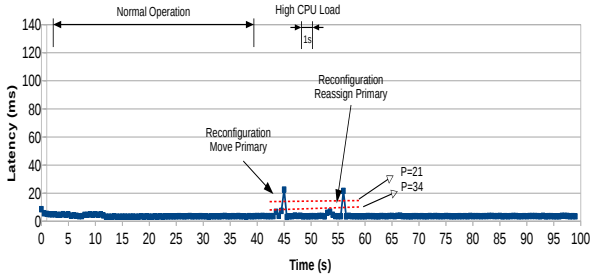
$$l_i = \frac{1}{5} * 100ms + \frac{4}{5} * \text{average r/t of successful ops over } i \tag{5.1}$$

We conducted experiments measuring P under different hot-spot durations for two different workload types. Figures 5.9, 5.10 and 5.11 depict executions for YCSB workload B (95%-5% reads/writes) on configuration C1 (Table 5.1) under hot-spots of duration 1, 2, 3 sec. Figures 5.12, 5.13 and 5.14 depict executions for YCSB workload A (50%-50% reads/writes) on configuration C2 (Table 5.1) under hot-spots of duration 1, 2, 3 sec. In Figures 5.12a, 5.13a, 5.14a no reconfiguration is taking place to mask the hot-spot, whereas in Figures 5.12b, 5.13b, 5.14b a reconfiguration is applied just prior to the hot-spot and after it. We report average YCSB latency at 500ms

of subsequent operations.



(a) Normal operation without reconfiguration



(b) Proactive reconfiguration and restoration

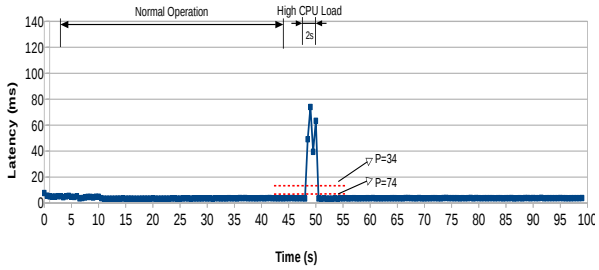
Figure 5.12: Mean operation latencies under a 1s hot spot (50% reads - 50% writes, Configuration C2))

intervals with the hot-spot occuring at t=50 sec. We calculate P with two different SLO targets, 5ms and 10ms, and report averages of P over 10 runs.

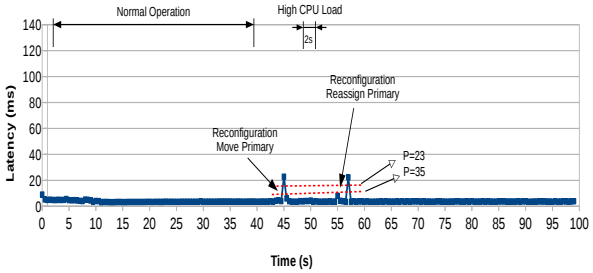
Average latency for both reads and writes under the 95%-5% workload is 2ms, increasing under the 50%-50% workload to 3ms due to more and costlier write operations. We observe that in the case of 95%-5% (Figures 5.9, 5.10, 5.11), reconfiguration has a higher cost that sustaining the hot-spot, across hot-spots of duration 1–3 sec. In the case of 50%-50% however (Figures 5.12, 5.13, 5.14) we observe the opposite for hot-spot durations of 2 and 3 sec, namely that the impact of reconfiguration is lower than that of the hot-spot (without reconfiguration). Thus, in this case it makes sense to decide to reconfigure rather than sustain the hot-spot. However, for a hot-spot that lasts 1 sec, sustaining the hot-spot is more cost-effective than reconfiguring around it (Figures 5.12).

To systematically compare the impact of reconfiguration around a hot-spot versus that of the hot-spot itself without reconfiguration, and determine the break-even point, we carry out systematic measurements of P under different configurations and hot-spots of increasing duration, depicting results in Figures 5.15, 5.16.

Figure 5.15 depicts the impact of a hot-spot of increasing duration on SLO on configuration C1 (Figure 5.15a) and C3 (Figure 5.15b) with 95%-5% reads/writes (YCSB workload B). The results depicted in Figure 5.15 are averages over eleven experiments with low standard deviation. In both cases, reconfiguration before and after a spike exhibits a near-constant cost, as expected, since it is independent of the duration of the hot-spot, whereas the cost of not reconfiguring increases with the duration of the hot-spot. Reconfiguration becomes cost-effective with an SLO target

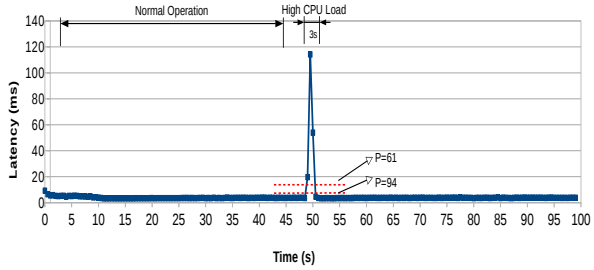


(a) Normal operation without reconfiguration

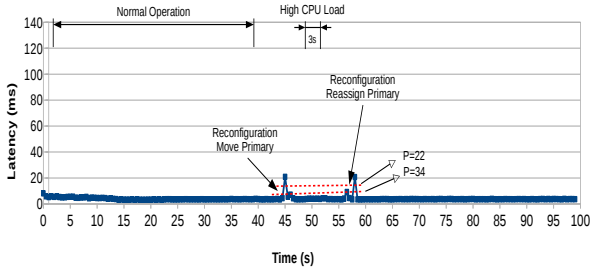


(b) Proactive reconfiguration and restoration

Figure 5.13: Mean operation latencies under a 2s hot spot (50% reads - 50% writes, Configuration C2))



(a) Normal operation without reconfiguration



(b) Proactive reconfiguration and restoration

Figure 5.14: Mean operation latencies under a 3s hot spot (50% reads - 50% writes, Configuration C2))

of 5ms for T $\geq$ 5 sec for 8 nodes and T $\geq$ 4 sec for 4 nodes. The lower cross-over point for 4 nodes can be explained as follows: With the same data set size being laid out on fewer shards, each shard is now twice the size compared to that with 8 nodes. As a result, a hot-spot has a stronger impact on overall performance for the same duration compared to Figure 5.15a, raising the penalty curve higher compared to 8 nodes. Thus it becomes cost-effective to use reconfiguration starting from shorter (about 3 seconds long upwards) hot-spots.

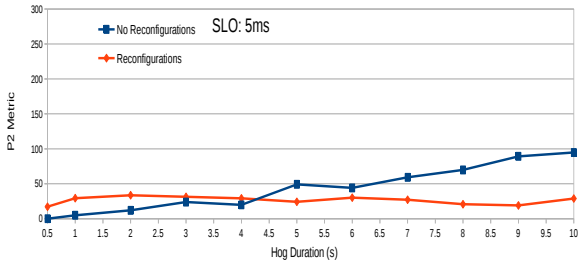
Next, we study the case where more nodes in each shard are involved in the completion of a request by setting up RethinkDB to require a majority of acks during a write (Configuration C2) and increase the proportion of writes in the mix to 50%-50% reads/writes (YCSB workload A). Our results indeed show higher response time (average of 3ms compared to about 2ms under 95%-5%) due to more costly write operations. We thus have set the SLO higher to 10ms in this case. Figure 5.16a de-

picts the results with the same dataset size showing that the cross-over point is lower (at about 1.5 sec) compared to previous experiments, making reconfiguration cost-effective for shorter hot-spots. As this is a heavier workload, even a short hot-spot results to large SLO violations, making the cost of reconfiguration equal to that of a hot-spot of the same duration (recall that reconfiguration lasts ≈100ms each way). To test the effect of a larger dataset that does not fully fit the node caches, we repeated the previous experiment with increased number of rows (6 million, Configuration C4) and depict the results in Figure 5.16b. The average latency in this case is at 3.5ms, increased by about 15% compared to the previous case. As such, hot-spots now have a stronger impact compared to the cost of reconfiguration (Figure 5.16b).

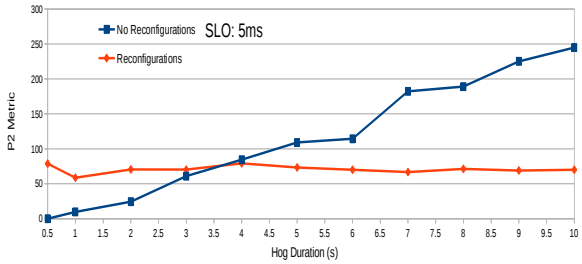
5.3.2 Simultaneous hot-spots across data-store cluster

We next examine policies under more complex hot-spot patterns. With operations involving a majority of nodes in a shard, reconfiguration should not help if a majority of nodes of a shard are hosted on nodes experiencing simultaneous hot-spots. Thus if hot-spots overlap in time, it may pay off to schedule them so that they never simultaneously execute on nodes that host replicas of the same shards.

To validate this hypothesis we perform experiments in which we randomly schedule two simultaneous 8-second hot-spots in an 8-node cluster using 50%-50% reads/writes (YCSB workload A) with Configuration C2. Prior to the hot-spots we apply reconfigurations aiming to mask both of them (two reconfigurations moving primaries away from the two impacted nodes, two additional reconfigurations to bring them back after the hot-spot is over), and compute the P penalty function for each execution with an SLO of 5ms. The initial placement of replicas is performed by RethinkDB (one shard primary per node, backups spread evenly across nodes, no two replicas of the same shard on the same node). Figure 5.17 depicts the P metric averaged over 8 repetitions of 30 YCSB runs, each randomly placing tw o hot-spots on the cluster and lasting for about 100 sec (500K ops). White bars correspond to cases where the random choice of hot-spot nodes yields no shard with two of its replicas on them. In these cases reconfigurations work as intended since by moving the primaries away, both shards can make progress with 2 out of 3 nodes in normally-loaded servers. In other experiments where some shard has two of its replicas on the impacted nodes,

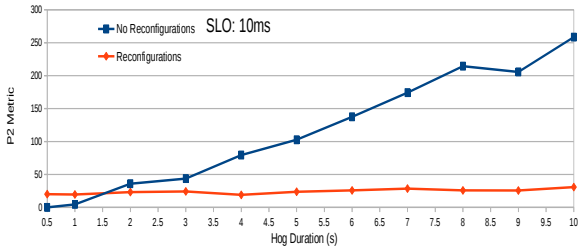


(a) Configuration C1

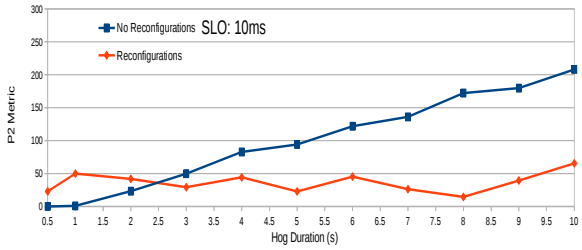


(b) Configuration C3

Figure 5.15: Impact of reconfiguration vs. that of hot-spot of increasing duration (95% reads - 5% writes)



(a) Configuration C2



(b) Configuration C4

Figure 5.16: Impact of reconfiguration vs. that of hot spot of increasing duration (50% reads - 50% writes)

reconfigurations are not helpful since even after reconfiguring the shard does not have a majority of normally-loaded nodes to make progress with.

Given that in Figure 5.17 we observe 21 out of 30 runs (70%) being impacted under two simultaneous hot-spots on 8 nodes, we expect that random placement of more simultaneous hot-spots will have even more drastic impact on cluster performance, with replica-group reconfiguration unable to mask their effect on affected shards. It is thus important for our reconfiguration controller to aim for scheduling the execution of hot-spots to avoid overlap, when such control is possible.

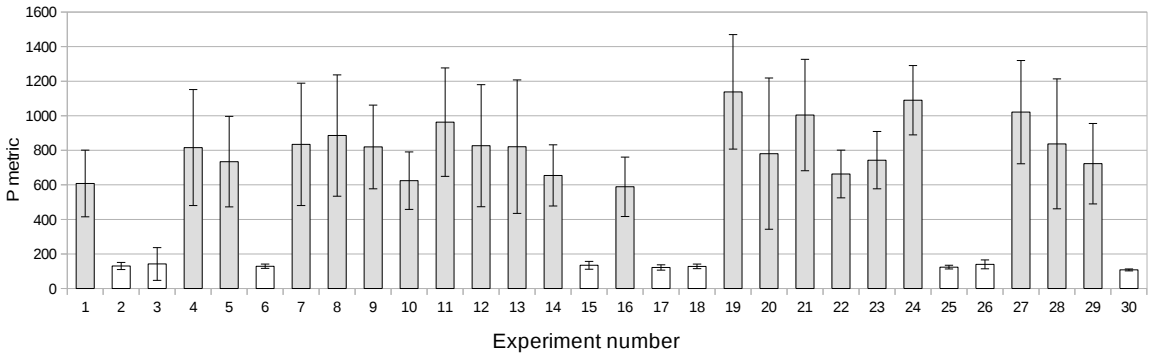


Figure 5.17: SLO violations when reconfiguring around two simultaneous 8 sec hot-spots on random servers (50% reads - 50% writes, Configuration C2). White bars correspond to cases where no shards have two of their replicas on the impacted servers.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we presented a cross-layer service management architecture for NoSQL data stores that leverages replica-group reconfigurations as a lightweight adaptation mechanism to reduce the impact of events such as node decommissioning or temporary hot-spots on data store performance. The architecture can utilize advance notifications about infrastructure activities provided by infrastructure management systems, functionality that has only recently been made available over standard virtualized infrastructures. In particular, in our prototype notification about nodes going down is provided by the Kubernetes container management system through the container-hooks notification mechanism, also available in other container management systems. Our prototype bridges between the Kubernetes container management system and containerized deployments of the RethinkDB NoSQL data store. Our evaluation on the Google Container Engine demonstrates that cross-layer management delivers the availability benefits of proactively handling the departure of a data-store server, as well as the performance benefits of masking the performance impact of a hot-spot through lightweight replica reconfigurations. Our experimental analysis and comparison of the cost of reconfiguring replica-groups to mask a temporary hot-spot

versus the cost of sustaining (without reconfiguration) the impact of the hot-spot shows that reconfigurations are cost-effective even for very short hot-spots (a few seconds), depending on the amount of load already placed on data-store servers. We have determined (theoretically and experimentally) that the effectiveness of reconfiguration actions diminishes with the number of hot-spots that simultaneously occur on the cluster. It is thus worthwhile –when possible– to defer (or otherwise schedule by actions of the reconfiguration controller) the occurrence of hot-spots so as to reduce their overlap in time.

6.2 Future work

An interesting direction for future work is to evaluate the benefits of the use of the replica-group reconfiguration management architecture developed in this thesis in a real setting, such as in masking the impact of hot spots produced by background activities in data-stores. Examples of such activities include asynchronous periodic snapshots, long garbage collections, data compaction / reorganization activities in data stores, or other ephemeral I/O or CPU-intensive activities that have a measurable impact on data store performance, especially under heavy application workloads. In such a study it would be interesting to evaluate scheduling policies that orchestrate hot-spot activities in terms of placement and time, to ensure that hot-spots have overall minimal impact on performance.

Bibliography

[1] "RethinkDB cluster management UI." https://www.rethinkdb.com/docs/sharding-and-replication/. Accessed: 2017-06-21.

[2] "Kubernetes." https://en.wikipedia.org/wiki/Kubernetes. Accessed: 2017-01-07.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (Stevenson, WA, USA), October 14-17, 2007.

[4] F. Gessert and N. Ritter, "Scalable data management: NoSQL data stores in research and practice," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, 2016.

[5] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI '00)*, (San Diego, California), 2000.

[6] M. Ghosh, W. Wang, G. Holla, and I. Gupta, "Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems," in *Proceedings of the 2015 IEEE International Conference on Autonomic Computing*, ICAC '15, (Washington, DC, USA), July 7-10, 2015.

[7] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris, "DBalancer: Distributed Load Balancing for NoSQL Data-stores," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), June 22-27, 2013.

[8] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *Proc. of the 7th International Conference on Autonomic Computing (ICAC'10)*, (Washington, DC, USA), 2010.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd ed.)," ch. The Primary-backup Approach, pp. 199–216, 1993.

[10] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, pp. 305–320, 2014.

[11] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. of 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN '11)*, pp. 245–256, 2011.

[12] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*, pp. 8–17, 1988.

[13] "MongoDB replication." `https://docs.mongodb.com/manual/replication/`. Accessed: 2017-01-07.

[14] "RethinkDB Architecture." `https://www.rethinkdb.com/docs/architecture/`. Accessed: 2017-01-07.

[15] P. Garefalakis, P. Papadopoulos, and K. Magoutis, "ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees," in *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, October 6-9, 2014*, (Nara, Japan).

[16] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, (Bordeaux, France), April 22-24, 2015.

[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, pp. 10:70–10:93, Jan. 2016.

[18] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (Prague, Czech Republic), April 14-17, 2013.

[19] "Kubernetes." `http://kubernetes.io/`. Accessed: 2017-01-07.

[20] "Amazon EC2 Container Service." `https://aws.amazon.com/ecs/`. Accessed: 2017-01-07.

[21] "Google Container Engine." `https://cloud.google.com/container-engine/`. Accessed: 2017-01-07.

[22] "IBM Bluemix Container Service." `https://www.ibm.com/cloud-computing/bluemix/containers`. Accessed: 2017-01-07.

[23] "RethinkDB." `https://www.rethinkdb.com/`. Accessed: 2017-01-07.

[24] "MongoDB." `https://www.mongodb.com/`. Accessed: 2017-01-07.

[25] "RethinkDB cluster management API." `https://rethinkdb.com/blog/1.16-release/`. Accessed: 2017-01-07.

[26] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving Large-scale Batch Computed Data with Project Voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, (San Jose, CA), February 14-17, 2012.

[27] "Basho Riak NoSQL database." `http://docs.basho.com/riak/kv/`. Accessed: 2017-01-07.

[28] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. N. Bessani, "FITCH: supporting adaptive replicated services in the cloud," in *Procedings of 13th IFIP International Conference Distributed Applications and Interoperable Systems (DAIS 2013)*, (Florence, Italy), June 3-5, 2013.

[29] I. Kitsos, A. Papaioannou, N. Tsikoudis, and K. Magoutis, "Adapting data-intensive workloads to generic allocation policies in cloud infrastructures," in *Proceedings of Network Operations and Management Symposium (NOMS)*, (Hawaii, USA), pp. 25–33, April 16-20, 2012.

[30] L. Wang, J. Xu, and M. Zhao, "Application-aware cross-layer virtual machine resource management," in *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, (San Jose, CA, USA), September 17-21, 2012.

[31] "Cloud Native Computing Foundation." https://www.cncf.io/. Accessed: 2017-01-07.

[32] "Autoscaling lifecycle hooks." http://docs.aws.amazon.com/autoscaling/latest/userguide/lifecycle-hooks.html. Accessed: 2017-01-07.

[33] M. Isard, "Autopilot: Automatic data center management," *SIGOPS Operating Systems Review*, vol. 41, pp. 60–67, Apr. 2007.

[34] K. Bhargavan, A. Gordon, T. Harris, and P. Toft, "The Rise and Rise of the Declarative Datacentre," Tech. Rep. MSR-TR-2008-61, Microsoft Research, May 2008.

[35] A. Papaioannou, D. Metallidis, and K. Magoutis, "Cross-layer management of distributed applications on multi-clouds," in *IFIP/IEEE International Symposium on Integrated Network Management, IM 2015, 11-15 May, 2015*, (Ottawa, ON, Canada), May 11-15, 2015.

[36] K. Magoutis, M. Devarakonda, N. Joukov, and N. G. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM J. Res. Dev.*, vol. 52, no. 4, 2008.

[37] "ReQL." https://www.rethinkdb.com/docs/introduction-to-reql/. Accessed: 2017-01-07.

[38] "RethinkDB changefeeds mechanism." https://rethinkdb.com/docs/changefeeds/. Accessed: 2017-06-21.

[39] "Docker." https://www.docker.com/. Accessed: 2017-06-21.

[40] "cgroups." https://www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/. Accessed: 2017-06-21.

[41] "etcd distributed key-value store." https://github.com/coreos/etcd. Accessed: 2017-01-07.

[42] "Kubernetes Horizontal Pod Autoscaling." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2017-06-21.

[43] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance differentiation for storage systems using adaptive control," *Trans. Storage*, vol. 1, pp. 457–480, Nov. 2005.

[44] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual storage devices with performance guarantees," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, (San Francisco, CA), 2003.

[45] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 14–28, 2014.

[46] "Kubernetes ReplicaSets." http://kubernetes.io/docs/user-guide/replicasets/. Accessed: 2017-01-07.

[47] "Kubernetes liveness probe." https://kubernetes.io/docs/tasks/. Accessed: 2017-01-07.

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (Indianapolis, Indiana, USA), June 10-11, 2010.

[49] "YCSB Core Workloads." https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Accessed: 2017-01-07.

Author's Publications

Research work in the context of this thesis resulted in the following publication:

- E. Bekas and K. Magoutis, "Cross-layer management of a containerized NoSQL data store," in IFIP/IEEE International Symposium on Integrated Network Management, IM 2017, 8-12 May, 2017, (Lisbon, Portugal), May 8-12, 2017.

Short Biography

Evdoxos Bekas is a graduate student at the M.Sc. program of the department of Computer Science and Engineering (CSE), University of Ioannina, Greece. Since 2016 he is a member of the Distributed Systems Group at CSE. He received his B.Sc. degree in Computer Science from the University of Ioannina in 2015. His research interests include distributed systems, cloud applications, and performance improvement of distributed applications.