NTNU – Trondheim
Norwegian University of
Science and Technology

Automatic scaling and maintenance of a NoSQL database

Eivind Siqveland Larsen
Knut Nygaard

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

NoSQL databases often support scalability and high availability. We address Voldemort, which is a popular, highly available NoSQL database that can be run on several nodes. Voldemort can be cumbersome to setup and maintain. As clusters grow in size, scaling into hundreds of nodes, management and administrative tasks become increasingly complex. We have therefore focused on automating management of a running cluster of nodes.

We have migrated the configuration storage of the Voldemort database from local XML files on disk to global objects using Apache ZooKeeper.

Using native tools and ZooKeeper coordination, we have implemented a fault tolerant, redundant management service. The service manages node discovery, configuration generation and propagation. It also has components for live monitoring and adjustment of responsibility to match each nodes available system resources.

Sammendrag

NoSQL databaser skalerer ofte bra, og tilbyr samtidig høy tilgjengelighet. Vi tar for oss Voldemort, en populær, høyt tilgjengelig NoSQL database som kan kjøre på flere noder. Voldemort kan derimot være tungvint å sette opp og drifte. Når clusteret vokser i størrelse, gjerne opp til hundrevis av noder, kan administrasjon og drift fort bli veldig komplisert. Vi har derfor fokusert på automasjon av administrasjon og drift i en samling av noder som kjører Voldemort.

Vi har flyttet konfigurasjon av Voldemort bort fra lokalt lagrede XML filer og over til globalt tilgjengelige objekter ved hjelp av Apache ZooKeeper.

Vi har brukt ZooKeepers egenskaper for koordinasjon og andre verktøy til å implementere en feiltolerant, redundant tjeneste for å ta hånd om nye noder, generere konfigurasjonsfiler, spre endringer og automatisk tilpasse det kjørende systemet etter hver enkelte nodes tilgjengelige systemressurser.

Acknowledgements

We would like to than Svein Erik Bratsberg for his guidance and assistance in the project. We would also like to thank LinkedIn for creating and publishing their take on Dynamo and making it open source. A big thank you also goes to Yahoo for publishing ZooKeeper. Without them this project would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

This thesis is split into 8 chapters. Chapter 1 introduces the project and our goals and includes a brief summery of our pre-project as well as early work with Voldemort. Chapter 2 covers technical background, related work on the topic of automated management and an introduction to Hbase and Cassandra - two distributed systems similar to Voldemort. Chapter 3 discusses Voldemort and ZooKeeper in detail. In chapter 4 we present our implementation and we introduce Headmaster, our stand alone automated management system. Chapter 5 present experiments we have run to test our implementation along with the corresponding results. Chapter 6 contains our evaluation of the results as well as our system as a whole. Finally a summary and conclusion is located in chapter 7. Our thoughts for further work can be found in chapter 8.

1.1 Introduction

This section expands upon the background for our project, what goals we have and what methods we have employed to reach our goals. Our end goal is a redundant, self balancing service for Voldemort that uses live data about a nodes health to make management decisions. By balancing we refer to the distribution of load among the nodes.

1.1.1 Background

Traditional relational database systems are generally very safe to use, usually providing all of the ACID (*Atomicity, Consistency, Isolation, Durability*) properties. This guarantees that all committed transactions are processed reliably. Today a lot of services needs to support up to millions of users and

serve thousands of requests per second. Experience has shown that databases providing ACID guarantees have trouble scaling. To allow for cost effective scaling, commodity hardware is used instead of expensive specialized servers. To continue scaling by adding numerous small servers, applications need to become increasingly distributed.

The main reason for ACID databases having troubles scaling is the strong guarantees of atomic operations, isolation and consistency. To allow for atomic operations in a distributed systems a distributed commit log would be required. Similarly to guarantee isolation distributed locks would be required. In a system with thousands of concurrent users lock contention can become a serious issue. Guaranteeing consistency across multiple machines requires significant overhead with regards to keeping all replicas consistent, and incurs a heavy cost on performance.

Distributed NoSQL databases often sacrifice consistency and isolation requirements to achieve higher availability with satisfactory durability. These systems often provide a highly available service and eventual consistency. The databases are designed to scale linearly, however managing and scaling these systems are not always trivial[7].

1.1.2 Voldemort and Dynamo

Voldemort is an open source distributed key-value database based off Amazon's paper on Dynamo, Amazon's highly available key-value store. Voldemort was created by LinkedIn and the first public release was in 2009. It is still under active development. Voldemort supports a simple set of operations limited to *put*, *get* and *delete*. Stored objects are uniquely identified by a key and are considered by the system as binary blobs. Voldemort is commonly used for storing lots of smaller objects, typically less than 1MB. As with other NoSQL implementations, Voldemort sacrifices consistency and isolation requirements to achieve higher availability with satisfactory durability. In fact, we will later see that most of this behavior is easily tunable and left as design choices per implementation. In section 3.1 we will look closer at the specifics of configuring Voldemort.

Compared with Apaches Cassandra, Voldemort lacks the power of column families and multi-key lookup. This means that any application powered by Voldemort requires additional logic to handle more advanced queries. For simple read heavy workloads however Voldemort is quite fast, reportedly serving over 20 000 read requests per second per node[6].

Voldemort is currently being used by a number of known companies. At LinkedIn they use Voldemort both as read-only and read-write stores. Services powered include LinkedIn Search, news and Who viewed your profile.

At EBay they use Voldemort as a read-only store for distributed lookups, and the dating site eHarmony uses Voldemort as a high volume read/write store.

1.1.3 Goals

Configuration of this distributed system is a complex and error prone task for system administrators. To simplify this configuration process, we want to move Voldemort's configuration data into ZooKeeper for easier coordination. In addition we would like to utilize the powerful features of ZooKeeper to create an automated service for managing a running Voldemort cluster.

We divide our goals into general goals and implementation goals. The general goals are as follows:

- Design a solution for simpler management and automatic scaling using Voldemort and ZooKeeper.
- Test and evaluate the suggested solution.

Our implementation goals will act as a road map for our project. They will be implemented in listed order:

1. Move Voldemort configuration data away from each local node and into the global domain of ZooKeeper
2. Integrate the Voldemort rebalance process with ZooKeeper
3. Create a service for automatic management of the cluster, including node discovery, membership and rebalancing with new members
4. Create a monitor service to monitor live nodes
5. Be able to act and make decisions about the cluster based on gathered information

While management and monitoring can be both useful and convenient, it does introduce overhead. This overhead should not have a significant impact on system performance, with Rabl et al. suggesting as low as 1-2% impact as tolerable[14].

1.1.4 Method

We plan to rewrite the MetadataStore in Voldemort to utilize ZooKeeper instead of local files. We will also create a separate service outside Voldemort for node discovery, membership and automatic management. For the nodes, we will create a status monitor service. The monitor service will act as decision support system for the automated management service.

1.2 There and back again

In this section we will try to shed some light on the preliminary work that was done before we ended up on the path that lead to our project. It consists of two parts. The first will be a summary of our pre-project and the second on our early work with Voldemort. This section can be skipped for most readers as it is not the most relevant for the thesis, but have been included because it was how the project started.

1.2.1 Energy efficiency of a Raspberry Pi cluster used for searching

Our pre-project was split into three parts. Building a cluster of Raspberry Pis, building a search engine and then measure performance and power efficiency of our cluster compared to a Core i5 Macbook Pro. During the build we constructed our own power supply and built the frame holding the cluster along with network equipment. The search engine was written in C to be as fast as possible. We used an existing python program to create an index over a corpus of text documents, and used tf-idf for scoring results. Through our performance testing we found that even though the Pis require a lot less energy to run, they are unable to keep up with modern hardware in a task so bound by CPU. It is worth mentioning that our entire index fit into memory, so a more disk bound application could have seen very different results. We however feel this is unlikely, due to the disk controller on the Pi being very slow and USB based with no DMA, so it requires CPU cycles to do any operation.

Early work with Raspberry Pis and Voldemort

In the early days of our project we were planning to use our Raspberry Pi cluster to host a Voldemort service. However the limited memory of 512MB quickly turned out to be a problem. The Pis could run Voldemort without any problems until some of the background data exchange processes started and killed the JVM. After some debugging we determined that memory was a serious issue, and abandoned the idea of running Voldemort on our Pis. They do not have enough RAM for any practical uses of the software.

To try out how much more powerful hardware you need to run a cluster based on Voldemort, we decided to give virtual machines a try. We used an older Mac mini (mid 2010) with a 2.4 Ghz Core 2 Duo processor and 8 GB of RAM, and created 4 virtual machines using VMware Fusion. They were all setup to run Arch Linux with 1 GB memory allocated. Initial tests

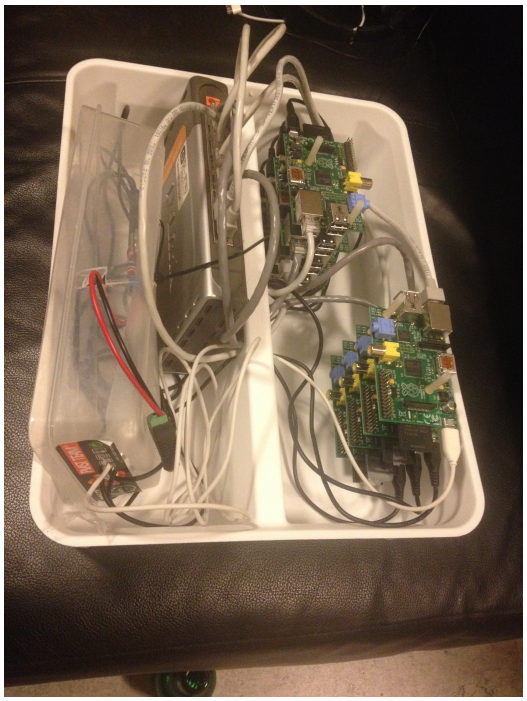


Figure 1.1: The Raspberry Pi cluster

quickly showed that these had no issues with running out of memory. After querying the institute for a proper solution we were allowed access to 4 virtual machines running on a server within the institute. These servers were running on a Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz host, with each VM granted 1 GB RAM. The network to these machines were however clogged by several 100mbit links, so we could not get much performance from them, and decided to work more locally.

1.2.2 Post mortem

In our pre-project we built a search engine and ran it on a cluster composed of 8 Raspberry Pis. In this project we wanted to work with a more developed distributed system and explore what possibilities existed for further work, but the typical software packages were unable to run reliably on a small system like the Pi. The main problem was the OS running out of memory.

Chapter 2

Background theory and related work

2.1 Technical background

The software used in this project employ many different and important techniques known from distributed computing, which will be discussed here. In section 2.1.1 we will talk about the key aspect consistent hashing. We will then discuss how consistent hashing techniques help us build *available* and *durable* databases.

Section 2.1.5 discusses how failures are handled in the system. The tunability of the database is introduced in section 2.1.6.

We lastly introduce PAXOS, a protocol for reaching consensus in a set of independent nodes.

2.1.1 Consistent hashing

Consistent hashing is a technique introduced in 1997 by David Karger[12] et al. The technique is used to solve problems with locating a key in a distributed system. Assume you have n servers in the system, then number these from 0 to $n-1$. The *hashspace* of the keys are then divided into n partitions. Then, to find which server to put a key on, do a

$$\text{Hash(key) mod } n$$

and you have the server number for the key.

However, with this scheme if a server fails and you now have $n-1=m$ servers, all the failed server's data is gone. You have to invalidate all existing servers, renumber your set of servers and start over. Analogously, if you add

k servers to accommodate higher traffic, you now have

$$\text{Hash(key) mod } n + k$$

and it is easy to see that all or most keys will have to relocate.

Consistent hashing reduces this problem by assigning each node with a number of points in the hash space. The hash space is viewed as a ring by wrapping around `0000-FFFF`. The nodes are placed at points along the ring as in Figure 2.1. When a key is looked up, you find it's place on the ring and move clockwise to the first point with an assigned node. In Figure 2.1, key K is routed to the next node on the ring, node B.

Now remember we let the nodes take on *several* points along the ring, thus spreading a nodes responsible key space to smaller pieces along the ring. I.e. nodes B, E and G in Figure 2.1 could be the same *physical* node.

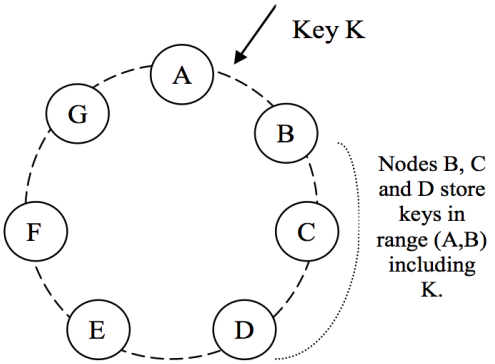


Figure 2.1: Hash ring with several nodes[9].

This approach to locating keys has several advantages:

- Like before, we distribute operations on the set of available servers. By having each server take numerous positions on the hash ring, we also gain a more even distribution of the amount of key space among the nodes, compared to giving them a single random or fixed position.
- Consistent hashing can also accommodate different types of servers by allowing more powerful nodes to have more points assigned on the ring, and vice versa.
- When adding a new server, a number of different areas around the circle are affected, and not one contiguous piece. This means the work of adding in the new node will be spread over the system, and not all requests routed to a single node.

- The spread of the key space also avoids rehashing of more than k/n keys when adding a new node.
- When a node fails, you move along the ring to the next server. When the nodes occupy several points on the ring, the extra work because of a failing node will be distributed among all the nodes in the system.

Consistent hashing also conveniently allows for easy, tunable replication of data to ensure durability, as used by key-value stores Dynamo[9] and Voldemort[6]. Consider W the number of data replicas to create. Hash the key, then move along the ring, writing key K to the W first nodes encountered on the ring. This will also ensure backups to be available along the ring when a node fails and requests gets passed along.

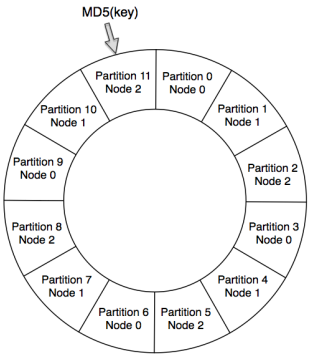
2.1.2 Durability

As explained in 2.1.1, both Voldemort and Dynamo use consistent hashing to locate and store keys. They are both key-value stores only. The goals for the Voldemort project is a database which is highly available and fault tolerant, providing an *always-on* experience.

To safely store user data, redundancy is required. The typical approach to this is synchronously storing replicas at different locations to ensure data replication. In practice though, providing this kind of strong consistency in a database system can prove detrimental to availability in some failure scenarios. A common approach to failure is locking down data and making it unavailable until the failure is resolved, so as not to serve potentially stale or erroneous data. While it is convenient for the programmer to never have to deal with possibly failed data, locking down doubtful data is not ideal for performance. In distributed systems network and system failures can be quite common, making guarantees of strong consistency very costly. In practice, strong consistency can be considered incompatible with high availability. This makes traditional replication methods unsuitable for highly concurrent distributed systems.

To help provide higher availability of data, an optimistic approach to replication is used. By allowing replicas to gradually propagate in the background and not synchronously, the workload on a distributed database system is greatly relieved, allowing for higher throughput. By also allowing conflicting data, we can continue to serve data in times of failure. This will cause more work for the application developers, and they need to be aware of this when using the database. Allowing conflicts also helps availability of put operations, and allows us the possibility to always accept a write.

It is easy to see how the consistent hashing ring helps implement this approach in Voldemort. To ensure data durability, one can simply write the data to the next few nodes on the ring. In Dynamo and Voldemort this is a tunable parameter, W, which controls how many nodes a write synchronously has to reach to be successful. Internally it is called *required writes*.



Partition hashed to	Preference list		
	N0	N1	N2
0	***0***	1	
1		***1***	2
2	3		***2***
3	***3***	4	
4		***4***	5
5	6		***5***
6	***6***	7	
7		***7***	8
8	9		***8***
9	***9***	10	
10		***10***	11
11	0		***11***

Figure 2.2: Hash ring with 3 nodes and 12 partitions from Voldemort[9] with N=2. Partitions are a fixed number of splits of the keyspace. Nodes are assigned a number of partitions which they hold. When a request is routed, one first hashes the key to find the partition the key belongs on. One then asks nodes in the order of the *preference list* for the key.

Similarly, it is easy to see how we can implement optimistic replication: By allowing nodes to push written objects in the background, at a later time, to any number of nodes further down the ring. This number N, is called the *replication factor*, and designates the number of replicas we *eventually* want to be present. I.e. we will have an *eventually consistent* form of replication.

You should also note that this replication has two benefits:

- As a data backup should the other node have a hard drive failure.
- As a functional backup. If the other node is off line, the replica can take over the workload with all the data available.

In Voldemort, the hash ring is split into X equally sized parts, called *partitions*. Each partition is assigned a unique ID and maps to a node. The nodes create a *preference list* over the partitions, which says to which *partitions* it should store the replicas.

Now let $N = 2$ with 12 partitions as in Figure 2.2. Let a key K's hash belong in partition 0. From the preference list, we see that this key should be put in `node0`, and replicated to the node holding partition 1, which is `node1`. Voldemort also does some sanity checking when replicating, such that replicas are always replicated to N *physical* nodes. I.e. In our example, if both partition 0 and 1 where held by `node0`, Voldemort would move along the ring until it finds the next partition that is assigned to a different node before storing the replica.

2.1.3 Dealing with data conflicts

Introducing optimistic replication to increase availability of the system, even in cases of nodes being unavailable for arbitrary reasons, has its price. In a distributed system nodes can be unreachable for a number of different reasons: Network splits, nodes dying, overloaded network points and concurrent operations (remember no isolation or locking) may introduce conflicts in the data set. These conflicts *will* arise and need to be discovered and resolved at some point.

How to deal with conflict is an important decision. The question of when to resolve them and who should have the responsibility of doing so all have different properties. A traditional approach is to resolve conflicts when they appear, i.e. on writes. This has the benefit of keeping read logic simple, but we may have to reject a write if not all (or a majority of) replicas are reachable. Resolving conflicts on write is also easy for the user to reason with.

Voldemort however wishes to run an always writable store. No data write should fail, even during periods of network splits and failing nodes. This requirement makes it impossible to resolve conflicts on write, and forces us to do conflict resolution on read.

This leaves the question of who should resolve the conflicts. We can either let the database store do conflict resolution or leave it to the application. If the store itself is going to resolve conflicts, it's choices in doing so are relatively limited. The store can not include domain knowledge for every possible application, so it can only do simple choices like last write wins or append the difference (which isn't even resolved, and probably destructive to the stored data). Recall that to the store most of the values are considered binary blobs.

The application however has intricate knowledge of the data and is in a much better position to make good decisions. Amazon[9] uses their shopping cart as an example. If two writes are two different products added to a shopping cart, we can record both of the writes even if they are divergent. Then at read, the application can use both versions to create a new cart with both products in it, i.e. merge the changes in a reasonable way, and present them to the user. This requires a bit of work from the developer, which they may not want or need to do, so there is also the option of doing simple conflict resolution on the server.

To be able to record all changes (writes), Voldemort uses a versioning system to record the history of all changes. The data are immutable blobs, and each write creates a new version with a history of predecessors. To record this version history, Voldemort employs *vector clocks*.

Roughly vector clocks are implemented as a list of `(node, counter)` pairs and this list is stored with every version of every stored object. By evaluating the vector clocks between two versions (or more), we can decide if one is an ancestor of the other or not. If `A` followed as a version of `B`, we can discard `B` and use `A` because the changes for `B` are already included in A. If the histories are divergent, we have a conflict and need to do some sort of merge or resolution. More formally an object `A` is an ancestor of `B` if all the clocks in `A` are less than or equal to the clocks in `B`. If so, we can safely discard `A`.

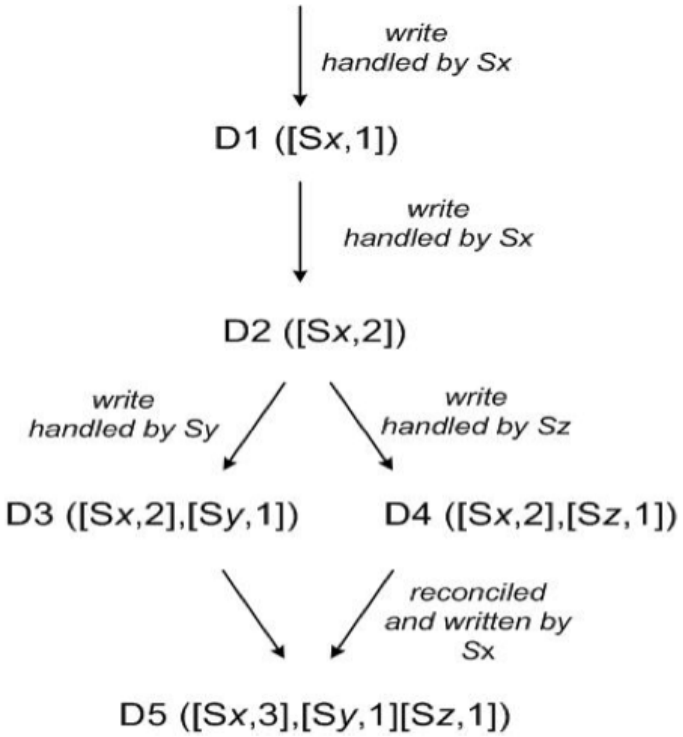


Figure 2.3: Version history for an object[6].

To use vector clocks in this manner safely, it is required that all update operations (writes) specify which version we are updating, or we would not be able to infer causality. This version information must first be obtained through a read of the object, and then passed along with the write. If during a read the database finds multiple divergent versions of the object, all of the objects at the *end* of the divergent paths are returned. The application then have to decide what to do, but typically a merged version has to be written. This merged version would specify the union of the versions of its predecessors as its version, creating a merged version.

An example of an objects version history is given in Figure 2.3. The different objects written are labeled DX and the vector clock associated with the object is given in parenthesis. The distinct nodes are designated with S followed by a character, for example S_x and S_y are two distinct nodes.

First a new object, D1 is written to the database. This write is handled by node S_x, and it assigns D1 the vector clock `([`S_x`,1])`. This is because it is the first version for this key and node S_x managed the write. If another object, D2, is written to the same key is managed by S_x, we increment the counter for the S_x version. We then have the object D2 `([`S_x`,2])`. Note that this means if somewhere we found D1, we could discard D1 if we are also holding D2 because of the versioning causality. D1 is the direct ancestor of D2, and is obsolete. Next in Figure 2.3, we have two more objects written to the key, but they are handled by different nodes. First a client reads D2, then writes object D3. The write of D3 is handled by node S_y, with the context from D2. We therefore get the object D3 `([`S_x`,2],[`S_y`,1])`. We can say D3 *decends* from D2.

Another client has also read D2 earlier. This client writes D4, which coincidentally is also written with the context of D2. D4 ends up being handled by S_z. D4 gets the vector clock `([`S_x`,2],[`S_z`,1])`.

As we can see, D3 and D4 are now on diverging paths, as both are based on D2. Now when a read happens, either D3 or D4 could be returned, but ideally, and at some point, *both* D3 and D4 will be returned. The application then needs to perform a *merge*, writing an object, D5, which includes the histories of both D3 and D4. We see that this indeed occurs, with D5 `([`S_x`,3],[`S_y`,1],[`S_z`,1])`.

To elaborate, we can see that we will never have to return D2 if we are following vector clocks. For both D3 and D4, the counter for S_x version for D2 $<= S_x$ for either D3 or D4. We can therefore discard D2 if we have either D3 or D4.

Over time, we can see that the list with vector clocks can grow significantly. Normally only the nodes in the preference list would write new versions of an object, which would be fine. But especially during network partitions, writes can be handled by many different servers not in the preference list for the target partition. According to Giuseppe DeCandia et al. (Dynamo)[9] they solve this with timestamps. A timestamp is stored with the individual version counters. When the number of versions exceeds a number N, the oldest vector clock is deleted. While this could remove important information and create issues in the database, Giuseppe DeCandia et al.[9] mentions that this problem has never occurred in production.

2.1.4 Membership

In a distributed system individual nodes sometimes need to know whichx other nodes are available in the system. We call this group membership. Whenever a node crashes we want its membership removed so that other nodes are aware of the crash. This can be achieved in various ways.

- A central service where nodes register themselves and send keep alive messages to verify that they are alive. When a new member is added or removed the service can simply broadcast a message to all members of the change. A drawback of this strategy is that we now have a single point of failure, as well as a solution that does not scale very well. As we expand and get more nodes in the system, the traffic into the central controller will increase as well. ZooKeeper can be used as one such central service, and we will look at how in the ZooKeeper section.

- The Gossip Protocol is a decentralized peer-to-peer solution to this problem. Gossip is modeled after how information spreads in social networks. By having nodes pair up with random partners at random intervals to exchange information, we allow information to spread throughout the network. Typically this is membership data, or information about "*who knows who*" and what their status is.

  For example if Alice discovers Charlie is down, when Bob calls Alice, Alice will tell Bob that Charlie is down.

  Gossiping thus allows the system to reach an eventual consistent state without having any *single* broad casted message or a demand for a single up to date catalog of the system state. To add a node to the group, simply pair it with one existing member and let the gossip protocol allow the news of the new node spread. Similarly if a node is unable to pair up with its partner the partner is marked as unreachable and this will be shared with future partners. This approach scales very efficiently as we only have peer to peer communication, but it is not without drawbacks. It is possible to create temporary logical partitions that exists while the information is being spread. Dynamo uses an external system with seed nodes to counteract these logical partitions.

  An example for adding a node with Gossip would be to tell, or *seed* a couple of nodes in the network with information about the newcomer manually. They will then gossip about this new node to their neighbors and the information will gradually spread. In practice this kind of seeding is useful because it really helps propagation go faster. Sometimes these seeding nodes can also be designated as such, and have a larger

role in the gossip. One might argue that introducing such hotspots for information is against the pure peer to peer nature of the gossip protocol, but Amazon regards it as necessary to speed up inclusion of new nodes.

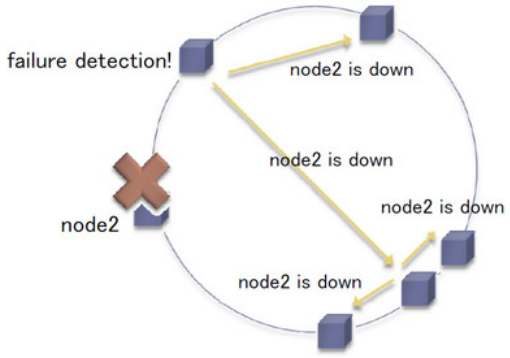


Figure 2.4: Gossiping about group membership. A node discovers `node2` is down, then relays this information to those in his neighbor list.

2.1.5 Failures

As we can see, these highly distributed systems are built to allow for failure. We will here explore how Voldemort and Dynamo try to mitigate for failing nodes while also keeping the operational requirements for consistency and durability.

Hinted handoff

In practice, both nodes and network will fail. During such failures, we still want to continue serving requests. To satisfy durability during failure, Voldemort perform a form of sloppy quorum. Instead of requiring all N members of the quorum to fulfill a durable write, they use temporary hosts. By this, we mean that in practice, the N nodes in the preference list a write eventually has to reach, are the next N *healthy* nodes on the ring. When a node on the preference list is down (marked as not healthy), the request is routed to the next healthy node. This write however contains meta data that says who was the intended target for this replica. The metadata is then used to push the write back to the intended target once he becomes healthy again.

Looking at Figure 2.2, if a key K hashes to partition 0, the preference list contains nodes `N0` and `N1` (recall N=2). If `N1` is not responsive, it is not healthy, and assume `N2` is the next healthy node. This means that when `N1`

is not responsive, N2 is placed on the preference list, because it is the next *healthy* node. Now when N0 receives a write, it will push the replica (to satisfy durability N=2) to N2 as long as N1 is down. N2 will however receive a write with a hint that this object actually belongs to N1, so N2 puts it in a special database to push it to N1 at a later time when N1 is healthy again. When N2 successfully reaches N1 and delivers the write, this object is deleted from N2. This process is called hinted handoff.

Using hinted handoff, we can still satisfy durability guarantees during node failures.

More failures

Hinted handoff helps in environments with low churn and where nodes usually come back online. But if a node holding a hinted replica goes down before the intended node comes back up, it may even be permanently down and the hinted object is never returned to the intended owner.

To mitigate such permanent failures, Dynamo[9] employs a method to regularly ensure that datasets common between nodes are consistent. A naive approach to this could be sending your entire key set to the other node and compare the data directly. This is however very costly, both in bandwidth used and is also CPU and disk intensive.

To minimize the impact on network load and node performance while simultaneously doing replica synchronization, the database uses hashes of the data. Here Voldemort and Dynamo have different approaches. In Voldemort, each key's hash is sent to the other node for comparison.

In Dynamo, Amazon have implemented merkle trees for comparing data sets between nodes. This significantly reduces workload on a replica synchronization. A merkle tree is a binary tree of hashes, see Figure 2.5, where the leaf nodes are hashes of the key objects. Now make a hash of the two children nodes. Then to make the parent node, assign the parent the hash of the two hashes of the children until you have the root node.

Each node keeps a merkle tree for the keyspace that it shares with the other node. The tree is updated whenever a write is commited. When a sync happens between two nodes, they only need to compare the value of their root node to know if they have any differences. One can also see how this method makes it fast to discover which keys differ. Each comparison will halve the search area. I.e. we have a binary search for the differing key or even subtree.

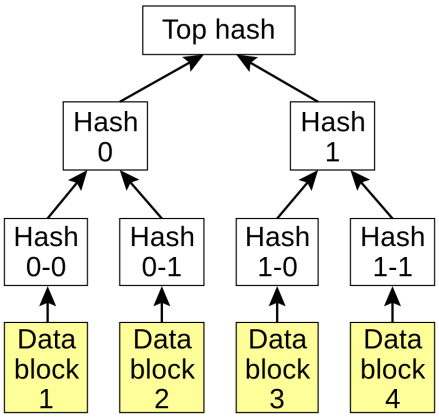


Figure 2.5: Structure of a merkle tree. Notice how the root node can be used for comparing two trees, and how the tree can be used for identifying divergent data.

2.1.6 Tunability

As we have briefly outlined above, the database system has three different parameters that are key to operation. The system gives the administrator the opportunity to fine tune certain trade-offs to implementation needs.

Internally, these are called N, R, W and controls certain aspects of the quorum that is involved in data operations. N is the *eventual* replication factor. This is the number of nodes we want to store replicas for a key on, and is allowed to happen asynchronously.

W is the write factor. This is the number of nodes that synchronously has to be part when we accept a write, and heavily affects performance. Typically W is set to two, to make sure an object put must have two replicas in the quorum to be considered successful.

Another interesting application is the highly writable property store we get with W=1. This means we would never reject a write as long as there is at least one node that can process a request. This will however jeopardize the durability of the data between returning the write and moving the object to other replicas. This setup also has a tendency to introduce a lot more inconsistencies in the data[9].

The parameter R controls how many nodes must be involved in a read for it to be successful. Setting R low (R=1) would give a high performance read engine. With R=1 though, we might end up with many inconsistencies in the data. It is the read quorum that must decide what data to return. During a read multiple versions of an object might occur. The quorum must

then reconcile their versions and return the latest, or return the conflicting ones. Because of this, discovering conflicts might be slower when only one node is involved in a read. We are in other words relying on only this single node's world view of the data in a optimistically consistent environment.

Knowing this, when executing a write, the first node to receive the put takes on the role as a coordinator for the write. The coordinator generates the new vector clock and writes a local copy. The coordinator then forwards the request to the top N healthy nodes in the preference list. Nodes that are not healthy are skipped. If W-1 of the healthy nodes report a successful write, the coordinator can return a successful request.

Reads are handled similarly. The first node receiving the request take on the role as coordinator. The coordinator sends the request to his top N other healthy nodes for the key. After R replies, the coordinator returns the latest version this quorum can agree upon. If there are any divergently versioned objects, all of them are returned to the client for reconciliation.

Amazon[9] mention that their most used configuration for Dynamo is a N=3, W=2, R=2 setup.

2.1.7 PAXOS

Whenever you have a distributed system where errors occur and messages get lost, even simple tasks like deciding on a value can become troublesome. PAXOS is a consensus algorithm made to solve this issue.

Say we have a set of processes that all can propose values. (The value could be related to leader election). Paxos ensures that a single one of these values is chosen and that all processes involved is notified of this value. If no value is being proposed, then no value should be chosen.

In Paxos a process can act in three different roles: *proposer, acceptor* and *learner*. A process is not limited to one role at the time. Processes communicate with one another by messages. These messages follow the non-Byzantine model and can take arbitrary long time to be delivered, or they can be lost entirely. They can also be duplicated, however they can not be corrupted. A process acting in a role might also crash, but we assume some critical information will remain through a restart.

When acting as a proposer, a process will propose a value to a majority subset of acceptors. These acceptors will then accept or deny the proposal according to some rules. Finally if a value is chosen, then all the learners will be informed of this value. The process of choosing a value is given below:

1. A proposer sends a prepare for proposal message to a majority set of acceptors with a number n. This number acts as a timestamp, and

represents the last(newest) message this proposer has seen.

2. Each acceptor receiving this prepare for proposal message will compare the proposed number n with their own highest observed n. If the proposed number is higher, then a promise of not accepting any proposal lower than n will be sent back to the proposer along with the acceptors highest observed value. If the proposers number is lower than the acceptors observed number, then the value is sent back along with the highest observed n.

3. If the proposer receives a promise from a majority of acceptors, it will send an accept message to the acceptors with the number n along with the value v, which is the maximum values received from the acceptors. If the number of promises is less than the majority of acceptors then the attempt has failed, and the proposer must start again with a higher number n.

4. When a acceptor receives an accept message, this acceptor will accept this proposal as long as there has not been issued a new proposal with a higher number in the meantime.

5. Finally, when an acceptor has accepted a proposal it informs all learners. Either by sending a message to all learners, or by sending a message to a designated learner, which in time will inform all other learners. This second step reduces message load but is less reliable.

It is possible to construct a scenario where two proposers keep out proposing each other and no progress is ever made. To combat this a distinguished proposer must be elected.

Paxos can typically be used in distributed systems to coordinate processes. One example is leader election, another is deciding on the next action to be taken.

2.2 Related software

In this section we will cover Cassandra and HBase. Two other distributed systems with similar applications and techniques as Voldemort. Lastly we will introduce SIGAR, a cross platform library for monitoring of system information that we used.

2.2.1 Cassandra

Cassandra is a scalable NoSQL database now available as open source through the Apache foundation. It was built from scratch with goal of being a massively scalable NoSQL database. In a cluster running Cassandra there is no sense of a master node, and all communication between nodes use peer to peer and the gossip protocol. As an administrator it is possible to scale Cassandra to meet both current and future system requirements.

Reads and writes

In Cassandra all participating nodes can be accessed for writes and reads. To ensure durability all writes are preceded by a write to a commit log. If a node has crashed, a node holding a replica will service reads and writes according to the hinted handoff strategy. As with Voldemort, Cassandra also features tunable consistency.

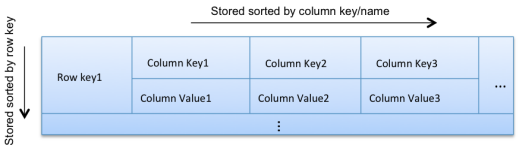


Figure 2.6: A sample Cassandra column[2]. The row key acts as a primary key and each key:value pair holds one field in a table.

Data model

The Cassandra data model is based on a key:value model, however Cassandra extends this model with up to two levels of nesting. This forms a map structure called columns where the outer row key acts as a primary key and the inner sorted map holds all information: `Map<RowKey, SortedMap<ColumnKey, ColumnValue>>`. By using maps we achieve easy lookups and range scans.

[2]Images taken from www.ebaytechblog.com

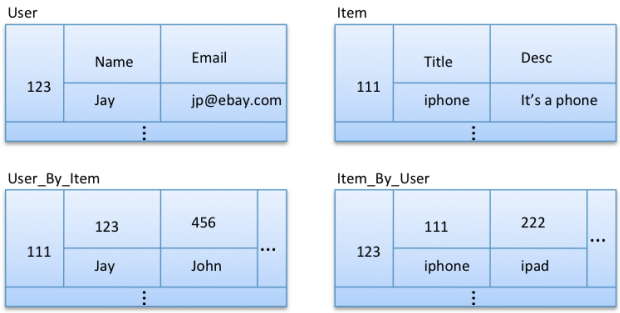


Figure 2.7: A sample Cassandra composite column. Here we have separate composite column to efficiently be able to serve queries on users by item and item by users.

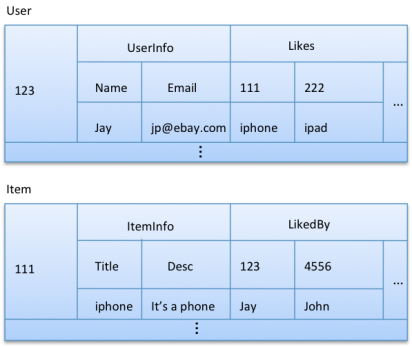


Figure 2.8: A sample Cassandra super column. Here we can see an outer map containing a person and the inner maps consisting of various composite columns

By adding one more level of nesting we can group columns. These are called super columns. In Cassandra de-normalization in important to efficiently perform queries that accesses information stored in separate columns. To allow for faster loopups composite columns can be created to match the need of one or more specific queries. These composite columns simply duplicate information already stored in various columns for more efficient access.

Cassandra is used by several well known companies. Netflix uses Cassandra for several applications, including its subscriber system and viewer history service. Facebook used Cassandra to power their inbox search, however this was abandoned in late 2010 in favor of HBase. Spotify also moved to Cassandra after migrating away from postgreSQL because of scaling issues. They use Cassandra to store playlists, radio stations and notification notifications.

2.2.2 HBase

HBase is based on Google's paper on BigTable. It is a sparse, distributed, persistent multi-dimensional sorted map.

With regards to the CAP-theorem (Consistency, Availability, Partitioning), HBase provides consistency and tolerance to partitioning, making HBase fault tolerant and easy to reason with in practice.

Important to note is that HBase provides a sparse multi-dimensional *sorted* map. Keys in the table are sorted, such that similar items are close to another when scanning a table.

E.g. when storing data about URLs, the keys are written in reverse: `com.google.www/users`. This scheme keeps URLs from the same domain and subdomain close to each other in the sorted map.

The *rows* or *entries* are typically sparse, meaning that most of the columns, in the table are empty.

To visualize a HBase record, it can help to think of a map of maps:

```
{
    "com.google.www/account": {
        "family": {
            "special": "value"
        },
        "anchor": {
            "com.cnn.www": "value"
        }
    }
    "com.google.www/users": {
        "family": {
            "": "value"
        },
        "anchor": {
            "no.nrk.www": "value"
        }
    }
}
```

Listing 2.1: A JSON approach to visualize how rows in HBase are structured. Idea courtesy of Jim R. Wilson[5].

The families for a record is static for the table at creation, but every family can hold any number of columns within, even many or none. This is where the sparse property comes from, as most entries will not have a value for the set of all sub-families in the table, leaving most column values empty.

Every *row* or *entry* also has an associated timestamp, used for versioning. This is typically a monotonically increasing number, such as seconds since

epoch. HBase stores a given number of versions of an entry, and can be queried for these. These different versions are stored in descending order, such that the highest number entry, i.e. the newest, is the first entry.

This gives a possible query structure as such `<key, family:column, timestamp>`. Timestamp is optional. If no timestamp is given, the newest entry is returned. If the query has a timestamp, the record with version equal to or less than the given timestamp is returned. If there is no such record, null is returned.

I.e. if we have a query `<no.nrk.www, referrals:no.nrkbeta.www, 999>`, and we have records in the database:

<no.nrk.www, referrals:no.nrkbeta.www, 1001>
<no.nrk.www, referrals:no.nrkbeta.www, 777>

`<no.nrk.www, referrals:no.nrkbeta.www, 777>` will be returned.

2.2.3 SIGAR

SIGAR (System Information Gatherer and Reporter)[15] is a cross-platform library and API for monitoring and gathering of system data. It is mainly written with java bindings in mind, providing a cross platform library that can be distributed as a single package. SIGAR also has bindings to many other popular languages, as the core is implemented in C, and accessed over Java JNI.

SIGAR provides a variety of features and information, translating it into operating system provided information where available. It can provide information about CPU, processes, memory, state, arguments, network interfaces, network routes and various system loads.

We use SIGAR as the low-level backend for collecting information on individual nodes.

2.3 Related work

In this section we will overview and discuss some related work to automatic scaling of distributed systems. Some are fully automatic, while others requires human intervention to execute.

2.3.1 Netflix Priam for Cassandra

Priam is a tool made open source by Netflix in 2012. It is a management tool that seeks to provide backup and recovery, automatic token assignment, a centralized system for configuration, and a RESTful interface for management and monitoring of Cassandra nodes. It is heavily integrated with the Amazon EC2 ecosystem (AWS), where Netflix keeps its clusters. Priam runs on every node that runs Cassandra, providing a novel interface to these services.

Backups use Cassandras built in snapshot tool to store SSTable snapshots in a (data center) local Amazon S3 bucket, which conveniently can be accessed everywhere. This SSTable backup is done daily. Incremental backups are then stored locally. Recall that SSTables are immutable, and it is easy to see this allows for efficient backup routines. Cassandra can also use these SSTables as a *base* when recovering a node, then use Cassandras internal mechanisms for conflict resolution to bring the new node fully up to speed. Netflix calls this an eventually consistent backup.

For scaling, Priam only allows doubling the size of the cluster. Priam does this by coupling existing nodes with a new node, then splitting the keyspace between them. This split of the keyspace for each node ensures a balanced ring when scaling up, so we don't end up with a imbalanced cluster that requires costly redistribution.

Priams token assignment (position in ring) can also be made aware of zones when distributing tokens. This is to ensure that there is at least one replica in each Amazon data center zone. In case of a network split, or connection issues in a data center, some replica will still be online at another one.

In summary, Priam has two very clear limitations with regards to scaling a cluster: the tool currently only runs on AWS (Amazon Web Services) and can only double the size of the cluster. Priam also makes sure to keep a cluster balanced when scaling.

2.3.2 Autoscaling Cassandra clusters

Baakind[2] created a system, called Hecuba, for automatic expansion of Cassandra clusters.

The goal of the project is an automatic scaler for Cassandra. The scaler should be able to increase or decrease the cluster size, based on resource usage in each node of the cluster. The scaler should not affect the performance of Cassandra. This includes the overall operational capacity for the cluster, as well as only executing scaling when the cluster can handle it, i.e. during times of lower load. The latter has not been implemented yet.

Hecuba includes a master service, which collects information from all the nodes in the cluster, and tries to keep a view of the current status. The Cassandra nodes periodically sends status reports to the master, e.g. one every second. The status report includes size of database, CPU load and memory usage. The master is configured with certain parameters for what loads are allowed, such as maximum memory and CPU load over *a given period of time*.

Like Netflix' Priam, Hecuba does scaling if a threshold is broken for a given period of time. Let's assume a CPU load of 60%, threshold 55%, and a period of 10 seconds. Hecuba will then only scale if the CPU load is reported as greater than *threshold* for at least 10 seconds in a row.

This setup leads Hecuba to heavily stressed nodes in the cluster, ie. Hecuba tries to identify hotspots to rebalance. In Cassandra, each node is responsible for a given token range (based on the hash of the key). If a node is under heavy load, Hecuba can split the nodes token range in half, assigning one half to the new node and keeping the other half. This split will hopefully halve the workload on the node.

A problem with this approach is that the data might end up being unevenly distributed. If we start with 4 partitions of the key range and 4 nodes. Assigning each node an equal part of the keyspace, we would ideally end up with 25% of the data on each node. Now if we split node 4, we might end up with 25% of the data on nodes 1-3, and 12.5% on node 4 and 5, creating an unbalanced cluster. This imbalance might eventually require a total rebalance, a very costly redistribution of the key space at a later stage. Hecuba scales the cluster by adding or deleting nodes one at time, so this fragmentation could quickly become a problem if not administered carefully.

Summarizing, the Autoscale system Hecuba scales a Cassandra cluster by identifying individual nodes that are under heavy load. This is done by having each node report their health to a given master. The master can then take action by splitting the token range for the node by adding a new node.

Chapter 3

Voldemort and ZooKeeper

In this chapter we will go into more detail with Voldemort and ZooKeeper. We will present Voldemorts various configuration data and have a closer look at the rebalance operation. With ZooKeeper we will explain key features as well as some common ways to use them.

3.1 Voldemort

In this section we will explore the essential configuration data used to manage Voldemort. We divide these configuration files into two groups: global and local files. Global files are identical files hosted by each individual node in a cluster, but they also need to be consistent between all nodes. These files contains persistent cluster metadata. As the number of nodes increase, managing these files can become cumbersome. Local files typically contain per-node specific persistent configuration data like hostname, ID and performance metrics.

3.1.1 Configuration data

Each node has at any point a full view of all the other nodes in the cluster and what partitions they hold. This is necessary to always directly route a key to the correct node. To provide this information, Voldemort distributes a XML file that describes the cluster, containing all hosts and partition information. This information is and needs to be consistent across all nodes. Note that the partitioning is strictly speaking decoupled from the storage, leaving (N,R,W) as a per *storage* setting.

Another feature of Voldemort is client side routing. Clients usually do routing of requests themselves, using the information of the cluster topol-

ogy to find correct node for a key. A client fetches the cluster config when connecting, then uses this information for routing. When the client starts receiving errors of wrong routing from one or several nodes, it rereads the config to receive the updated information.

In our running cluster we use the following global configuration files:

- `cluster.xml` contains persistent information on all nodes involved in a cluster. All nodes must have an entry in this file before they can join the cluster. The fields in the file is relatively self explanatory. In addition to listing communication ports, each entry must have a unique ID and hostname. The partition field tells each node which parts of the keyspace a node is responsible for. In Voldemort each partition can be moved between nodes, however the total number of partitions is not changeable after the cluster is running. In the example below the reader can verify that we have 16 partitions spread across 3 nodes. It is also possible to assign nodes into zones. Zones are typically used when operating in multiple data centers limiting traffic between them.

```
<cluster>
  <name>ntnucluster</name>
  <server>
    <id>0</id>
    <host>voldemort1.idi.ntnu.no</host>
    <http-port>8081</http-port>
    <socket-port>6666</socket-port>
    <admin-port>6667</admin-port>
    <rest-port>8085</rest-port>
    <partitions>4, 3, 8, 0, 13, 11</partitions>
  </server>
  <server>
    <id>1</id>
    <host>voldemort2.idi.ntnu.no</host>
    <http-port>8081</http-port>
    <socket-port>6666</socket-port>
    <admin-port>6667</admin-port>
    <rest-port>8085</rest-port>
    <partitions>7, 1, 6, 15, 9</partitions>
  </server>
  <server>
    <id>2</id>
    <host>voldemort3.idi.ntnu.no</host>
    <http-port>8081</http-port>
    <socket-port>6666</socket-port>
    <admin-port>6667</admin-port>
    <rest-port>8085</rest-port>
    <partitions>12, 2, 10, 14, 5</partitions>
  </server>
```

```
</cluster>
```

Listing 3.1: Sample cluster.xml

- `stores.xml` contains persistent metadata on all stores operated by a cluster. One cluster running Voldemort can have several pluggable stores, served by different backends. These backends can even potentially be other databases, like a mysql or BDB database. As with `cluster.xml`, each node must have it's own copy of this file. The parameters in this file control each store's behavior.

  The persistence field defines what kind of backend storage is used. Voldemort supports BDB, mySQL, memory, read-only, and cache. Cache and memory are both implementations that reside only in memory, the difference being how they react when out of storage space. It is possible to have multiple stores using different backend technology.

  Routing-strategy defines how replicas are stored in the cluster. Voldemort offers three alternative strategies: consistent routing, zone routing and all-routing. When using consistent routing all requests will be routed to the first N nodes from the keys location in the consistent hashing ring. Here N is the replication factor. Zone routing sits on top of consistent routing and ensures that each request goes to zone-local replicas. This is used to limit traffic across data centers. It also allows for zone aware replication where we want replicas in different data centers. Finally All-routing simply routes the request to all nodes specified by the call.

  As with routing, we also have 3 alternatives for hinted handoff strategy: any-handoff, consistent handoff and proximity handoff. When using any-handoff, a random live node in the cluster is selected for the request. With consistent hand-off enabled, one of the N nodes on the hash ring after the failed node will handle the request. Finally with proximity handoff requests will be routed according to the zone proximity of the clients zone id. This is especially useful if an entire data center is offline.

  We can also finely tune each individual store with regards to availability, consistency and durability. Replication factor specifies how many duplicates we want for each entry in the database. Required read and writes specifies how many nodes must respond to a request before it is considered successful. These three parameters greatly influences performance.

Finally it is possible to specify the format on keys and values. Supported key formats are: json, java-serialization, string, protobuff, thrift and identity. Values can have the same formats and can be compressed with gzip or lzf.

```
<stores>
    <store>
        <name>test</name>
        <persistence>bdb</persistence>
        <description>Test store</description>
        <owners>harryhogwarts.edu,
           hermionehogwarts.edu</owners>
        <routing-strategy>consistent-routing</routing-
           strategy>
        <routing>client</routing>
        <hinted-handoff-strategy>consistent-handoff</hinted-
           handoff-strategy>
        <replication-factor>2</replication-factor>
        <required-reads>1</required-reads>
        <required-writes>1</required-writes>
        <key-serializer>
            <type>string</type>
        </key-serializer>
        <value-serializer>
            <type>string</type>
        </value-serializer>
    </store>
</stores>
```

Listing 3.2: Sample stores.xml

- `server.properties` is used to configure individual nodes in a cluster. The node ID must map to an entry in `cluster.xml` for it to be valid. Using the maximum threads field one can tune the application running to the hardware of the individual node. This config file also contains store related login infomation used to access those.

```
node.id=0
max.threads=100

############### DB options ######################
http.enable=true
socket.enable=true

# BDB
bdb.write.transactions=false
bdb.flush.transactions=false
```

```
bdb.cache.size=2G
bdb.one.env.per.store=true

# Mysql
mysql.host=localhost
mysql.port=1521
mysql.user=root
mysql.password=3306
mysql.database=test

#NIO connector settings.
enable.nio.connector=true
request.format=vp3
storage.configs=voldemort.store.bdb.BdbStorageConfiguration,
voldemort.store.readonly
ReadOnlyStorageConfiguration
```

Listing 3.3: Sample server.properties

- We also have several local configuration files used during a rebalance operation. `rebalancing.steal.info.key` contains information on which partitions the node will need to fetch from other nodes in the system. The rebalance process also stores the existing `cluster.xml` and `stores.xml` before starting the rebalance operation in case of a roll back. Finally there is a `server.state` file that is used to persistently store rebalance state in case of a failure. It is either `NORMAL_SERVER` or `REBALANCING_MASTER_SERVER`.

3.1.2 Rebalancing

In this section we will discuss how rebalancing is done in Voldemort. A rebalance operation involves repartitioning and moving data between nodes. There can be several reasons for rebalancing a cluster including adding nodes (cluster expansion), adding nodes to zones(zone expansion) and load balancing by shuffling partitions around. We would also like to be able to shrink the cluster (cluster contraction), however this is not yet supported by Voldemort. During a rebalance Voldemort suffers no down time and clients should not experience any noticeable performance drops. In addition there should be no loss or corruption of data if anything goes wrong and the rebalance is aborted. A rebalance is split into three steps which we will explain below.

Terminology

Before we can go into details about the rebalance process we need to define a few terms:

- Stealer-node: When rebalancing this node will *steal* a partition from another node and copy its data.
- Donor-node: When rebalancing this node will act as a donor and send data to one or more stealer nodes.
- Donor-stealer pair: During a rebalance donors and stealers form pairs and copy data while proxying requests.
- Task: A list of all patition-stores that must be cloned from the donor node.
- Zone n-ary: Closely tied to how many replicas there is of each data item. If we have a replication factor of 3 then the original entry is the Zone 0-ary, the first duplicate the zone 1-ary and the last one zone 2-ary. This information is used when deciding stealer-donor relationships during a rebalance.
- Proxy-bridge: When moving partitions during a rebalance, stealer-nodes are responsible for data they do not yet hold. To solve this, proxy bridges are put up between donor and stealer nodes so any request can be forwarded to a node holding the data.

Preparation

Before a rebalance can start, we need a plan. This plan contains which partitions to move to which nodes. When choosing where to move partitions, Voldemort has a set of design principles to assist in deciding where each node should copy its data from during a rebalance:

1. Only steal if you do not already host the required partitions
2. Try to steal from a node in the same zone, if this is not possible, steal from a donor in the same zone as the primary partition.
3. When you have to steal, steal from the same zone n-ary.

Following these principles the plan generated will limit data movement across zones in addition to aligning proxy-bridges with stealer-donor pairs. The output of such a planning operation is a file called `final_cluster.xml`. This is a modified version of the `cluster.xml` and contains the newly proposed cluster setup. Planning is done by running one of the rebalance scripts provided with Voldemort.

```
./bin/rebalance-cluster-expansion.sh -c current_cluster -s
    current_stores -i interim_cluster -o output dir
```

Listing 3.4: Sample command to plan a cluster expansion. Outputs a `final_cluster.xml` as well as a plan

Execution

The first issue that needs to be handled when rebalancing is dealing with already connected clients. When a client connects it requests cluster metadata from Voldemort. The client uses this metadata for routing all future requests. When rebalancing, this metadata is no longer valid and clients must be informed. Voldemort has two ways of dealing with this issue. Either the client can check metadata version on each request or nodes can alert the client if they receive a request for a key they do not have responsibility for. This alert is implemented by throwing an InvalidMetadataException. In the latter case clients only refresh their metadata after requesting a partition that has moved.

Voldemort offers a proxy-pause option while rebalancing which is a window of time where the updated cluster metadata is propagated, and proxy-bridges are set up. This allows connected clients to request the new metadata before any actual rebalancing is done. The rebalance is executed using the supplied administrator tool.

The steps involved in executing a rebalance are the following:

1. Upload interim metadata: To allow for a new node to join the cluster it must have an entry in `cluster.xml`. This is done by replacing `cluster.xml` with a modified `interim_cluster.xml`. In this file an entry for the new node has been added. This node will not yet be responsible for any partitions.

2. Gather verification data: To later be able to verify that the rebalance was successful it is possible to extract some verification data for comparison before running the rebalance.

3. Stop asynchronous tasks: These tasks could cause the rebalance to abort.

4. Execute rebalance: Run the appropriate rebalance script.

```
./bin/run-class.sh voldemort.tools.RebalanceControllerCLI --url
    \$URL  --final-cluster final-cluster.xml --parallelism 8 --
    proxy-pause 900
```

Listing 3.5: Sample command to execute the rebalance. Parallelism defines how many tasks can be run at the same time.

Servers will be notified of the rebalance by the client sending the new `cluster.xml` files and changing their state to REBALANCING_MASTER_SERVER. Servers will now start proxying requests according to the new partition setup. During the proxy-pause interval, nodes will pair up in donor-stealer pairs based on tasks and start proxying requests as needed. After the proxy-pause has ended, donors will start to send data to the stealers. Once all task has been completed proxy-bridges are torn down, server state changes back to NORMAL_SERVER, and the cluster returns to normal operation.

Verification

After a rebalance is completed, it can be useful to verify that the rebalance was a success. There is also a repair job that should be run after the rebalance to delete any orphaned keys. Recall that the entries shared by a donor node are now orphans, and can be cleaned up by deleting them.

```
./bin/run-class.sh voldemort.utils.KeySamplerCLI --url
    $BOOTSTRAP_URL --out-dir key-samples --records-per-partition
    5
./bin/run-class.sh voldemort.utils.KeyVersionFetcherCLI --url
    $URL --in-dir key-samples --out-dir key-version-fetches
```

Listing 3.6: Commands for pulling a key sample from a store and the versioned data objects stored under the keys. These are used to verify that the rebalance did not corrupt data

```
./bin/voldemort-admin-tool.sh --repair-job --node '-1' --url \
    $URL
```

Listing 3.7: Sample repair job script. Passing -1 as node will run the script on all nodes.

Recovery from failure

A benefit of having a strict key-value store, is that there are no advanced features to consider when moving data. No invariants that must be kept or avoided, no joins, foreign keys, searches or other indexes (except for the primary, the key) that complicates moving and partitioning of data. Think of Voldemort as a giant, distributed and persistent hash map. This simplifies the process for performing rebalancing, i.e. scaling a lot. We can simply copy the partition, with some precautions to ensure durability and integrity.

When scaling, we give a list of $(node, partition) \rightarrow node$ pairs. The left side is the FROM node and right side TO, the destination node. A list of such pairs is called a *rebalance plan*, a set of transfers that are to be executed during the rebalance. The first part of a secure transfer, is the setup of proxy bridges. This starts the routing of request to the *new* layout. ie. if we are moving $(node0, 3) \rightarrow node1$, all requests that are going to partition 3, are now routed to the new node, $node1$.

After allowing proxy settings to propagate for an interval, the receiving node, the *stealer*, initiates a request to transfer the target partition from the *donor*. Now to allow for a safe transfer, all PUTs to node1 are also sent (proxyed) back to node0. This ensures no writes are lost in case of an abort or other failure. All GETs are first looked up locally in case they already have been transferred. If not, the key is fetched explicitly from the donor and returned. In case of failures, we abort the plan and set the routing back to the old configuration.

It is easy to see that this set of rules allows a partition move to be canceled or crash at any time without incurring data loss.

3.2 Apache ZooKeeper

We have relied heavily on ZooKeeper in order to store our Voldemort configuration data as well as handling coordination in our management system. In this section we will explain what ZooKeeper is along with how it works. Finally we will present which techniques we have used along with how we used them.

3.2.1 What is ZooKeeper

From Apache ZooKeepers own website[1] they explain ZooKeeper as:

> ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

ZooKeeper was designed without specific primitives (i.e. locks and leader election). The reason being that they didn't want to tie down the developer, and instead expose a simple jet powerful API. This allows developers to create their own primitives based on their application specific needs. This means ZooKeeper can be adapted to suit the requirements of different applications.

3.2.2 Clarification

A `write` or `put` in the context of ZooKeeper is considered a ZooKeeper `setData` operation.

A ZooKeeper event, is an event triggered by a watch set in ZooKeeper, giving push information to a client about a change in a ZooKeeper znode.

In the context of ZooKeeper, the word file might be used for a znode. File used in the context of a local filesystem is a local file.

3.2.3 Znodes

ZooKeeper behaves almost like a file system with one key difference. Instead of files and directories - we have znodes. All znodes can contain information in the form of bytes. One znodes can have children which again can have more children. Together they form an hierarchical structure, addressable like a unix filesystem path.

Znodes also include version numbers for data changes, ACL changes and timestamps to allow cache validations as well as coordinate updates. Whenever a znodes data is changed, it's version number increases.

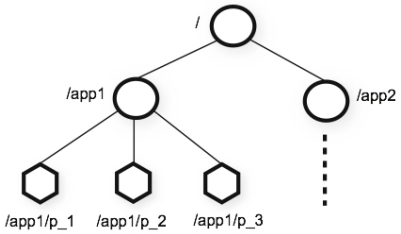


Figure 3.1: A sample namespace

These znodes can have different flags set which control their behavior. These are *persistent*, *ephemeral* and *sequential*. Persistent is rather self explanatory. When this flag is set during a PUT operation, the data will remain in ZooKeeper even though the client disconnects. Ephemeral is the opposite. Data placed in ZooKeeper with this flag will be deleted when the client ends its session. If sequential is set, each child of the znode will have a monotonic increasing sequence number appended to its path. This number is relative to the parent znode, so children znodes can have equal or different names. Ephemeral and sequential are powerful features which enable us to utilize ZooKeeper for leader election, coordination as well as locks.

3.2.4 ZooKeeper Guarantees

ZooKeeper provides two basic ordering guarantees:

- Asynchronous linearizable writes
- FIFO client order

Since we have asynchronous writes a client can have several outstanding operations, but ZooKeeper guarantees that they will be executed in FIFO order. ZooKeeper also provides clients with the option to listen for changes on a znode. ZooKeeper guarantees that a client will receive a changeEvent before seeing the new state of the system following the change.

3.2.5 Watches

As mentioned above, a client can request a watch on a znode in ZooKeeper. This watch will give the client a changeEvent the next time this znode is modified. This could be changes in number of children, changes in the data or that the znode has been deleted. These watches can also be put on non

existing znodes which gives the client a changeEvent when the znode is registered in ZooKeeper. It is however important to note that a watch will only trigger once, and it is not possible to request infinite watches. So it is up to each individual client to request a new watch after receiving a changeEvent.

Because of ZooKeepers linearizable guarantees a read following one of these watchEvents will always return the new state of the system. In the event of several changes in quick succession it is possible for a client to not be notified of each individual change. It will however always be able to determine the final state of the system after all the changes.

3.2.6 Implementation

As mentioned ZooKeeper can be used for more than just storing configuration data. In this section we will explain some different ways to use ZooKeeper.

Group membership

ZooKeeper can be used to keep track of group membership. To do this we create a znode for the group at a given path. Now members can register their membership by creating a child znode with the *ephemeral* flag set. As long as they have a session with ZooKeeper they will be listed as a member of the group. When a member registers as a child they also put a watch on the parent node. If there is a change in membership, for example when a node is added, then all nodes will receive the changeEvent.

Note that only a message of *change* is delivered. This means that if 3 nodes quickly joins, one watch is delivered for the first change. The client must then query for a new children list. Because of ZooKeepers FIFO ordering guarantees, this request will either include all changes, or the second and third write will trigger a new watch. In effect we will never have the case that when a node is added, and a process watching is not notified.

Leader election

Leader election is something that frequently needs to be addressed in a distributed system. In our own management service we used ZooKeeper to handle this issue.

To elect a leader we utilize the *sequential* and *ephemeral* flags of a znode. We have a znode called leaders in ZooKeeper where potential leaders can register themselves. This znode has the sequential flag set so we get a total order on the potential leaders. We also force each potential leader to use the ephemeral flag when registering. This ensures that when a client disconnects,

it is removed from the children list. When it rejoins, it is given a new sequence number. By default the znode with the lowest sequence number will be the leader.

If a node registers as a potential leader but does not win the election (i.e: There is another contender with a lower sequence number) it puts a watch on the winner and waits for changes. If eventually the leader disconnects then there will once again be elected a new leader based on sequence numbers.

Any client wanting to get a hold of its leader can simply ask ZooKeeper for all the children of the parent znode, and then determine which one is the leader based on sequence numbers. Now the client can put a watch on this leader znode and be notified whenever there is a change in leadership. If said event is received the client simply ask ZooKeeper again for the children and repeat the process.

This however can cause a thundering horde effect. When the leader disappears, ZooKeeper suddenly have lots of watches to deliver, as well as a horde of incoming requests for children lists of the parent node. To avoid this, every node waiting to be the leader can listen on the *previous* node in the queue. Because of the linearizable guarantees, we know that we will never become the leader until the one in front disappears. This way only one watchEvent has to be delivered, causing only one request for the updated znode list.

Locks

Another use case for ZooKeeper is managing access to resources. This is implemented with locks. To create a lock we have a reserved znode for the resource in ZooKeeper. To queue for the lock, clients must create *sequential* and *ephemeral* children on this node. The client with the lowest number holds the lock. We use ephemeral to prevent a client for keeping the lock forever in case of a disconnect, and sequential is for ordering of clients.

It is also possible to separate locks into read and write locks. In ZooKeeper their only difference will be in the naming of the znode, but the client now can regard them as read or write locks. The case explained above is a typical write, or exclusive lock. In the case of a read lock, the client will be allowed access to the resource as long as there is no write locks ahead of it. If there is a write lock ahead of the read, the client will put a watch on the previous exclusive lock and wait until it is deleted. Such read/write locks can typically be created by prepending a “r” or an “x” on the client znode, to signal a read or exclusive lock, respectively.

Chapter 4

Integrating Voldemort and ZooKeeper

In this chapter we will cover our implementation process. It is structured in 4 parts. Part 1 consists of integrating ZooKeeper support and mechanisms into the Voldemort project, making Voldemort read and store configurations in ZooKeeper. Part 2 contains our work towards rebalancing with the help of ZooKeeper. In part 3 we create the service Headmaster, which is a management service to run along side of the Voldemort. Headmaster keeps track of active nodes in the cluster and generates configuration for new nodes that are added at runtime. It also includes them in the current cluster and executes rebalances to transfer data to the new nodes. We have also created SigarAgent and StatusAnalyzer to allow for Headmaster to make management decisions based on recent node activity levels. All our work and code can be found in our git repository[3].

Lastly we have included a part 4 on how we wrote tests to validate our implementation during development.

4.1 Part 1

Integrating ZooKeeper into Voldemort and taking advantage of the coordination services ZooKeeper provides, proved rather difficult. We chose to override and rewrite much of the current MetadataStore implementation to one that uses ZooKeeper as the backend storage engine instead of local files. Actual data is still of course stored locally, but configuration (metadata) now resides in the ZooKeeper cluster. This abstraction works rather well for shared data files, but causes some issues with information that is stored for individual nodes. As such, we landed on a fixed path scheme, where certain

configuration files are expected to reside on known locations, and will be presented later.

Using ZooKeeper as storage backend introduces a dependency of a running ZooKeeper cluster for normal operation of any node. While this is a liability and potential source for downtime in a highly available service, ZooKeeper is itself a highly available system and the 5 node setup is regarded as very stable and in heavy use at Yahoo[10]. In any case, the system will in general be able to operate through short ZooKeeper outages, as all meta and config data is cached locally, and will not affect operation.

4.1.1 Configuration

Configuration is stored in a VoldemortConfig object. Normally this is created from settings stored in local files per node. At startup, the server looks for a properties file in a directory given by environment variable `VOLDEMORT_HOME`.

Now, if you specify a ZooKeeper connection URL as a command line parameter, the program looks for configuration in ZooKeeper. We have given an example outline of the expected directory structure in Figure 4.1. This is for the three nodes `vold0.idi.ntnu.no`, `vold1.idi.ntnu.no` and `vold2.idi.ntnu.no`. You can see the expected path follows the pattern `/config/nodes/HOSTNAME/server.properties`.

The startup code then reads and tries to parse the config file. If parsing fails, the server will fail. This is to clearly signal that something is wrong during startup, and needs immediate fixing. If the file is not found, the server registers a watch on the path in ZooKeeper. If there is a write to this file, ZooKeeper will let the server know and the config file can be re-read.

One can argue whether to fail or listen when a config is not found, but we decided for the latter, to listen for changes, after implementing the management service. As we later will see, listening allows us to push new config files to a node by the help of our management daemon Headmaster.

Our modification inherits from the original VoldemortConfig object, so that the ZooKeeper capable version can be used in the same code, in the same manner. The modified code reads data from a fixed location in ZooKeeper, and sets the necessary settings and properties for the server to boot and join the cluster. The ZooKeeper connection URL is passed as a command line parameter.

The directory structure for the whole program can be viewed in its entirety in Figure 4.2, and has been included for completeness.

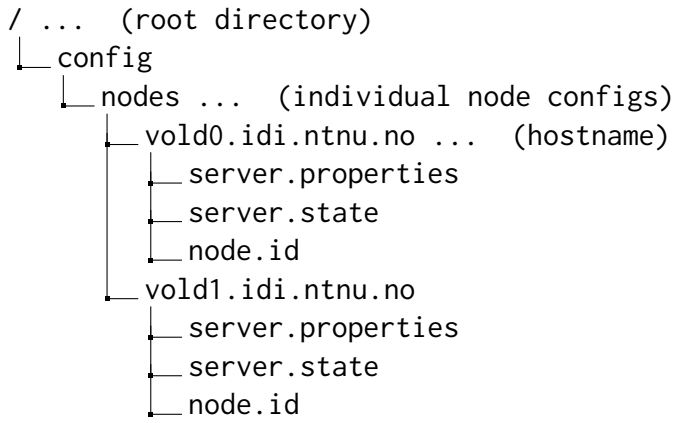


Figure 4.1: Individual node configs in the shared space.

4.1.2 MetadataStore

The internal state for a node is written to a `Store` called `MetadataStore`. It consists of an in memory cache store, and a persistent one on disk. The different stores are implementations of an abstract data `Store`. As such, we wrote a `Store` using ZooKeeper as the backend, and used this for persistent storage.

At startup we read the global configuration, the cluster and stores settings, from the Store. The Store fetches the requested files from ZooKeeper, and leaves a watch on the nodes. Because a store *only stores* data, the Store itself can not take advantage of the event deliveries from ZooKeeper: The Store keeps no application logic, and can be thought of as a hash map.

To notify and update the internal state of the node on events, we found it necessary to receive events in the configuration management logic, written in the `MetadataStore`.

When an event from the ZooKeeper client is delivered to the `MetadataStore`, we first sort it by type. If it is a management event, like a disconnect event, we temporarily disable the ZooKeeper backend and serve data from the cache until it is reconnected. In our testing, such disconnects (session expiry) happened about once every hour, but the reconnect was usually done in less than two seconds. This instability may be due to our ZooKeeper cluster consisting of a single node and being on a different network.

After a session expiry, we receive a reconnected event when the ZooKeeper client reestablishes a connection. At this point, all set watches are lost, and all events during the connection loss will have been lost. We must therefore reset all watches and check the global configuration data on reconnect.

```
/ ...  (The root can also be a chroot.)
├── active
│   ├── vold0.idi.ntnu.no ...  (ephemeral)
│   └── vold2.idi.ntnu.no ...  (ephemeral)
├── config ...  (persistent)
│   ├── nodes (individual node configs)
│   │   ├── vold0.idi.ntnu.no
│   │   │   ├── server.properties
│   │   │   ├── server.state
│   │   │   └── node.id
│   │   ├── vold1.idi.ntnu.no
│   │   │   ├── server.properties
│   │   │   ├── server.state
│   │   │   └── node.id
│   │   └── vold2.idi.ntnu.no
│   │       ├── server.properties
│   │       ├── server.state
│   │       └── node.id
│   ├── cluster.xml
│   └── stores.xml
└── headmaster
    ├── headmaster_0009 ...  (ephemeral & sequential)
    ├── headmaster_0011 ...  (ephemeral & sequential)
    └── headmaster_0012 ...  (ephemeral & sequential)
```

Figure 4.2: Directory structure in ZooKeeper. The nodes or directories are actually znodes and can hold data.

4.1.3 Handling the ZooKeeper connection

Our way of handling of the ZooKeeper client and connection went through many stages and changes as we got a better understanding of its features.

Our currently preferred design can be viewed in `ActiveNodeZKListener`. The ActiveNodeZKListener class wraps a ZooKeeper client connection, providing event delivery to the interface `ZKDataListener`. This approach removes a lot of the noise generated by the client, and delivers clear events that are simpler and clearer to handle and reason with when coding application logic.

We provide the following listener events:

dataChanged(path): This method is called when data has been written to the watched znode on `path`.

nodeDeleted(path): A call to the listener to let it know the znode on `path` has been deleted.

reconnected(): The connection handler provides a simple call `reconnected()` when a connection is reestablished after session expiry. When this call happens, we know it is safe to do sanity checking of our state and register watches again.

childrenList(path): This one is a little bit special. This method is called whenever a znodes childrenList is changed. Recall that znodes are like nodes in a tree, and can have many children. This method call indicates new or removed children for the node on `path`.

We also include the possibility for receiving the raw events from the ZooKeeper client in the interface:

process(event): `Event` is the raw WatchedEvent from the ZooKeeper client. Registering as a `Watcher` will forward every internal ZooKeeper event to the listener.

4.2 Part 2

4.2.1 Including rebalance functionality in Headmaster

Voldemort uses external scripts for triggering rebalancing and repartitioning. This made incorporating this functionality into Headmaster very cumbersome. We included two classes into the project: `RebalancePlannerZK` and `RebalancerZK`. These expose the methods: `createRebalacePlan()` and `rebalance(RebalancePlan plan)` used to trigger these events. Both classes utilize `ActiveZooKeeperListener` to fetch required configuration data from ZooKeeper. Headmaster now have access to these methods and can trigger them if deemed necessary.

4.2.2 Recursive triggered writes of meta data

While adding the rebalance features, we bumped into an issue related to the architecture of Voldemort. In a rebalance, the new config files are pushed (written) to each node separately using Voldemort admin data requests. This causes every node to execute a PUT of the new data on the MetadataStore.

As explained above, we use ZooKeeper for (persistently) sharing the global configuration files. Also, we would like to be notified about changes to the config files using watches. So first you have N nodes in the cluster,

executing N writes to the same file. Each file write potentially triggers N watches, causing a read and watch reset. Worst case we will end up with N writes and N*N watch triggers and reads, all in quick succession (less than a second). We therefore decided to ignore such put requests in the ZooKeeper driven persistent Store, and only put the new data in the Metadata cache store, deferring the admin to use a ZooKeeper write operation to put the new config out, making the nodes do a re-read.

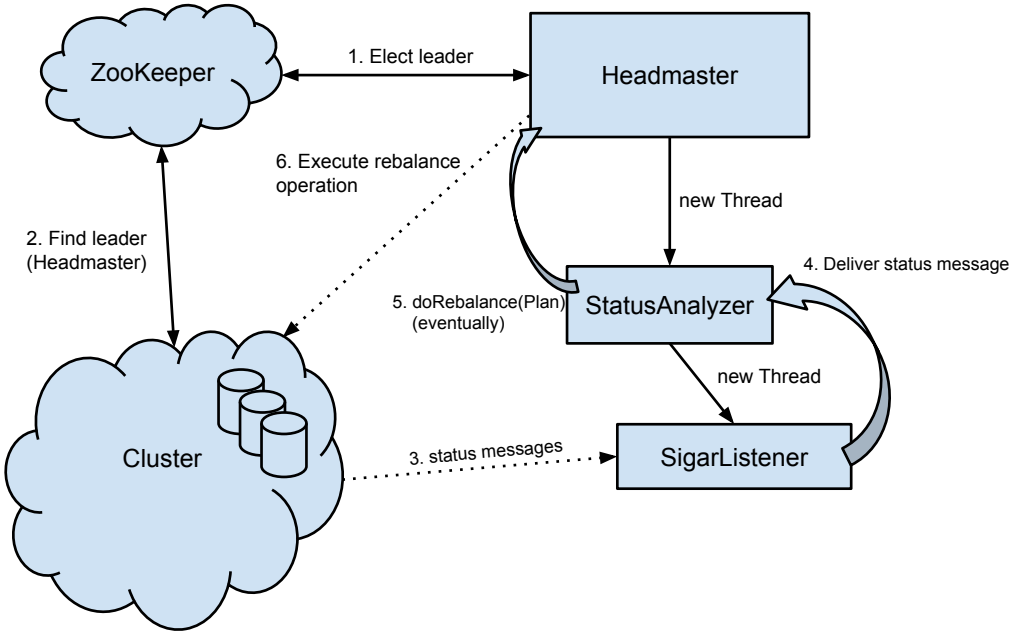


Figure 4.3: System drawing of the management service Headmaster.

4.3 Part 3

4.3.1 Headmaster

Headmaster is our clusters management service. It can be started on any node as long as ZooKeeper is reachable. When an instance of Headmaster is started an *ephemeral* and *sequential* znode is created on the path /headmaster in ZooKeeper. This means that each instance will automatically have its own sequence number appended to its name. So at runtime with 3 nodes running Headmaster we typically have a structure like in Figure 4.2

Headmaster can be running on multiple nodes concurrently, and a leader election protocol based on ZooKeeper is responsible for only having one active

Headmaster at the time. If the current Headmaster becomes unavailable, one of the waiting Headmasters will take over and become active.

Whenever a node is starting Voldemort, the node registers itself under a znode called `/active` in ZooKeeper. This is done with the *ephemeral* flag set. When a node creates this znode it also put its node id as the value. If the node has not yet received an id, it sets the value to "NEW".

Through ZooKeeper, Headmaster is informed of all changes on the `/active` path. So, whenever there is a node registering, Headmaster checks for the value "NEW". Headmaster then checks if the node has an entry in the current `cluster.xml`. If the node has an entry, Headmaster simply changes the value to the correct id. If there is no entry in `cluster.xml`, Headmaster assigns the node a unique id and creates a znode under `/config/nodes/` and uploads the corresponding `server.properties` file. Finally `cluster.xml` is updated with a new entry for the new node. This node will not have responsibility for any partitions yet. A rebalance to give the newcomer a part of the keyspace can be triggered manually or by StatusAnalyzer.

4.3.2 SigarAgent

SigarAgent is our service for monitoring activity on nodes in the cluster. Our solution is heavily inspired by the work of Andreas Baakind[2]. We utilize Hyperic's sigar API to gather system information on each individual node and send this to Headmaster. We continuously send status on current CPU usage, memory used and how much data the node has stored compared to a given maximum. In contrast to Baakind who only informed his scaling service if any value was above a set threshold, we send a status message every 5 seconds. To prevent these messages for taking up too much resources, we use UDP instead of TCP as we do not require all messages to arrive at Headmaster. During all our testing we had 0 lost packages.

SigarAgent utilizes ZooKeeper to keep track of who is current Headmaster, and will react if there is a change of leadership. If there is no Headmaster active, SigarAgent will sleep until it is notified of a new Headmaster.

4.3.3 StatusAnalyzer

StatusAnalyzer is a decision system available to the current Headmaster. StatusAnalyzer receives status messages from all nodes running SigarAgent and stores them in a `HashMap<String, LinkedList<StatusMessage>>`. We store the last 10 messages received from each node, corresponding to 10 intervals of data. This HashMap is used to analyze the cluster state and trigger rebalancing actions if deemed necessary.

StatusAnalyzer has a few useful member functions:

```
getAvgCPU(String hostname)
getCalmestNode()
analyzeCPU()
migratePartitions(List<String> strugglingNodes)
```

Listing 4.1: Helper functions in StatusAnalyzer.

Every 30 seconds analyzeCPU is run to see if the average CPU-utilization of any node is over a threshold. We have defined a node running at over 84% as a struggling node. If any nodes are above this threshold, getCalmestNode is run to see if there are any calm nodes . Any node with less than 70% load is defined as a calm node. If any node is struggling and there is a calm node, migratePartitions is triggered. This will inform Headmaster that we need a rebalance action to stabilize the cluster.

We have two strategies for moving partitions. If we have more than one struggling node, we pick a node at random and move one partition over to the calm node, or we move one partition from each struggling node over to the calm node. The former strategy will ensure a slow and steady transition while the latter could arrive at a stable state faster at the cost of performance during the more heavy rebalance. Moving several partitions at once could lead to a calm node being overwhelmed.

Figure 4.3 gives an overview of the monitor/decision cycle and the typical ordering of events.

4.4 Part 4

4.4.1 Testing

When working on large projects, a lot of code lines are involved. To prevent breaking functionality underway, and to help verify correct (desired) behavior while working, we decided to write unit tests. Because of the complex series of events that need to happen while running the program to execute your desired code, it is much easier and more efficient to test and verify your code programmatically in a test environment. The design and verification of our Headmaster program greatly benefited from this approach. The event chain to test the code can be fairly complex, even though the application logic is fairly straight forward.

To create a test environment, we used JUnit and the Mockito project to mimic the behavior we need for validation. Mockito is a testing framework for Java. It allows to create "mock" objects that can be used to trigger and verify program behavior in a predictable and controlled manner. The "fake", but controllable objects are very useful for eliminating outer dependencies, and instead having them behave in a completely predictable and controlled way.

Listing 4.2 shows a short example of how Mockito and JUnit is used to write short, simple yet powerful functional tests that verify behavior. It is also worth noting that the resulting test code is quite readable by a programmer.

```
@Mock
ActiveNodeZKListener activeNodeZKListener;

Headmaster headmaster;

public void whenDataChangedTestIfWatchIsReset() {
    String path = "/config/cluster.xml";

    when(activeNodeZKListener.getStringFromZooKeeper(
        path, true)).thenReturn(EXAMPLE_CLUSTER);

    headmaster.setZKListener(activeNodeZKListener);

    headmaster.dataChanged(path);

    /**
     * verify method is called with params path, and
       watch flag set to true.
     */
    verify(activeNodeZKListener,
       times(1)).getStringFromZooKeeper(path, true);
}
```

Listing 4.2: Test code utilizing Mockito. Think of the `@Mock` class as a subclass with all methods overrided `return null;`.

Here we are verifying that upon receiving a notification that a znode has changed, the znode (path) is fetched with a new watch set. All without actually providing a working third party object (ActiveNodeZKListener) to the test, the object is entirely faked with the static `when` and `thenReturn`

methods. After setting the dependency class up with the desired behavior, it is injected into the class we are testing, then verify that the desired method is called once and only once (`times(1)`). Similarly it is easy to see how you can verify two calls (`times(2)`). For zero calls, it is recommended to use the more readable method `never`.

Chapter 5

Experiments

In this chapter we will present benchmarks we have run and their corresponding results. We will benchmark both the original Voldemort implementation and our own to compare the impact of our work on the overall throughput and performance of the system. We will look at requests processed per second, response time distribution and CPU-utilization under a typical usage pattern.

Our experiments are done in two phases. First part is a single-node benchmark of each node that will participate in the tests. This is to get an understanding of their performance. As our cluster consists of very different hardware it will be useful to have an idea of what to expect from each individual node before we add them together. The second part concerns performance and scaling after and during rebalancing of a cluster. The rebalance process will be executed both manually and using our automatic system for balancing.

The goal of these experiments is to verify that after our modifications the performance remains intact. For Voldemort, near linear scaling of throughput is expected when adding worker nodes to the cluster.

5.1 Setup

We will here present the setup and tools used for our testing.

5.1.1 Benchmark tool

We use Voldemorts provided benchmark tool to send data to our cluster. The program is heavily based on the work by Yahoo Cloud Serving Benchmark[8]. This allows us to generate different types of workloads, request loads and

number of sending threads. We also had to do some modifications to get a running average throughput, instead of a total average throughput when measuring.

We parse most of the logs in python to transform the data points into something useful. In some cases we also have to manually put the data together. For analysis, we mainly use `R`[13] with extensive use of the `Hmisc`[11] library. Our Rscripts can be found in our project on github[4]. All the diagrams in this chapter has been generated with `R`.

An example run with the benchmark tool could be:

```
read_percent=95
write_percent=5
metric=histogram # [summary|histogram]
ops=1 000 000 # total requests to run
threads=54
recordcount=1 000 000 # records to insert during warmup
valuesize=1024 # bytes per record

voldemort-performance-tool.sh --threads $threads --metric-type
$metric --ops-count $ops --url $url --value-size $valuesize
--store-name $store --record-count $recordcount $@
```

These are also the settings we used for most of our tests.

5.1.2 Hardware

We have 4 computers involved in these experiments:

1. Desktop computer: Intel Core i7 3.5 GHz Quad Core 16 GB Memory 250GB SSD (load generator)

2. MacBook Pro: Intel Core i7 2.6 GHz Quad Core 8 GB Memory 250GB SSD

3. MacBook Pro: Intel Core i5 2.5 GHz Dual Core 8 GB Memory 120GB SSD

4. Mac Mini: Intel Core 2 Duo 2.4 GHz Dual Core 8 GB Memory 250GB SSD

In addition we use a D-Link Dir-655 gigabit router for networking. All computers are equipped with a single gigabit ethernet connection.

5.1.3 Software

All computers run Apple OS X 10.9.2 operating system and Java JDK 1.7.0_51.

5.2 Single-node benchmarking

To verify our modifications are not too costly for performance, we will be concentrating on two metrics during our tests. The response time distribution has been included to verify *how fast* we are answering most queries. The throughput metric has been included to verify whether raw, overall work performance is intact, ie. *how many* queries we can answer.

In this part we want to investigate how our computer and cluster behaves in various scenarios. We will in all experiments use a workload of 95% read and 5% write requests as is common on a typical social website where most browse with a few comments or status updates. Each data entry is 1kB. The database is seeded with 1 million entries before each test starts, unless otherwise specified. In all tests the i7 desktop is generating requests.

5.2.1 Response time distribution

Response time is an important metric and the response time distribution is very helpful in determining perceived performance. Response time is important for both user experience when using a service and for upholding SLAs.

We want to establish a baseline for how each individual node performs with regards to time used responding per request. This is to verify the impact our modifications have on the users experience of performance. In this experiment we will setup a single node Voldemort instance, and measure the service time over a given number of requests. We will run this test with both our ZooKeeper implementation and the official Voldemort code for comparison. Queries are generated as fast as possible to test response time at maximum load, i.e. the worst case.

We expect comparable results between the nodes, as all nodes will run at maximum capacity. We also expect the majority of these requests to be serviced in under 1 ms as found by Rabl et al.[14].

Throughput of single nodes

We also want to get an overview over each node's ability to serve requests. To test throughput we will again run single node clusters and send requests

at maximum load. The throughput is measured over 1 million requests, for values of 1024 bytes.

As our hardware is so different we expect vast difference in individual node performance. The old core 2 duo will not fare well compared to the i5 and i7.

5.2.2 Adaptive cluster

In these experiments we have a look at how a cluster consisting of several nodes performs. We will explore different scenarios both with and without rebalancing, and look at how throughput and CPU usage is affected. For these experiments we increase the preloaded values in the system to 3 million to get a bigger data set to move during each rebalance operation.

Baseline

This experiment will act as a baseline for how a cluster behaves under load. We have 2 nodes sharing 16 partitions, 8 partitions each. We run this experiment on the two slowest nodes to make increases in performance clearer, as the other two nodes limits are beyond the capability of our load generator. As we only have 2 nodes in the cluster we only use one required read / write on each request and no data duplication. Recall that this is a `(1,1,1)` system if we refer to the (N,R,W) parameters of Voldemort.

We expect both nodes to struggle with servicing their workload and as a result the throughput should be less than optimal. In other words we do not expect the throughput of these two nodes combined to be equal to the sum of their individual throughputs, i.e. we expect to experience sub-linear scaling.

Manual rebalancing

In this experiment we want to see how the cluster behaves when we allow a struggling node to move partitions over to a node under less stress. We will manually move one partition at the time over from a struggling node to one less worked. The cluster will consist of the Core 2 and the i5, the two slowest machines. We will use the same criteria as in our automatic system, which means that a struggling node is defined as a node running at 85% utilized CPU or higher. We will continue to move partitions until there are no nodes over 85% utilization. This experiment is run on the original Voldemort code.

During rebalancing we expect the throughput to suffer somewhat, however we expect to see an increase in performance over time as we gain a more

optimal distribution of partitions.

Automatic rebalancing

This experiment is a rerun of the previous one except there is no manual interaction and Headmaster, our management service, is executing all operations based on monitor data.

We expect this experiment to play out the same way as the previous one, with comparable performance. We expect the performance to be comparable as we did not observe any additional costs introduced by our code in the earlier tests.

Automatic cluster expansion

In this test we will utilize all our features by adding a node to an already struggling cluster and automatically move partitions to alleviate the existing nodes. We will have a cluster consisting of the Mac Mini and i5 MacBook Pro and we will add the i7 Macbook Pro during the experiment. Headmaster will be in charge of triggering rebalance as needed.

We expect the system to perform poorly during the initial phase as both computers will be running at near max capacity. After we add the third computer and move the first partition, the performance should increase. As the system migrates more partitions away from the struggling nodes the overall performance should keep increasing.

Large scale move of partitions

Finally we want to investigate how much throughput suffers during a rebalance. We will increase the size of the data set to 15 million records to have a longer rebalance stage. This will also make sure that the entire work set does not fit in memory. We run this as a stand alone test because the other tests moved data too quickly to yield enough data points for proper analysis. We will run this experiment with a cluster consisting of 16 and 32 partitions to see if there is a relationship between partition size and performance during a rebalance. In both experiments the nodes will host the same amount of keys, but spread over varying numbers of partitions.

As the entire work set does not fit in memory, we expect throughput to be lower than in the other tests. During a rebalance, we expect the throughput be consistent but lower than before the rebalance started.

5.3 Results

In this section we present our results from the experiments. The experiments follow the same order as they were presented.

5.3.1 Response time distribution

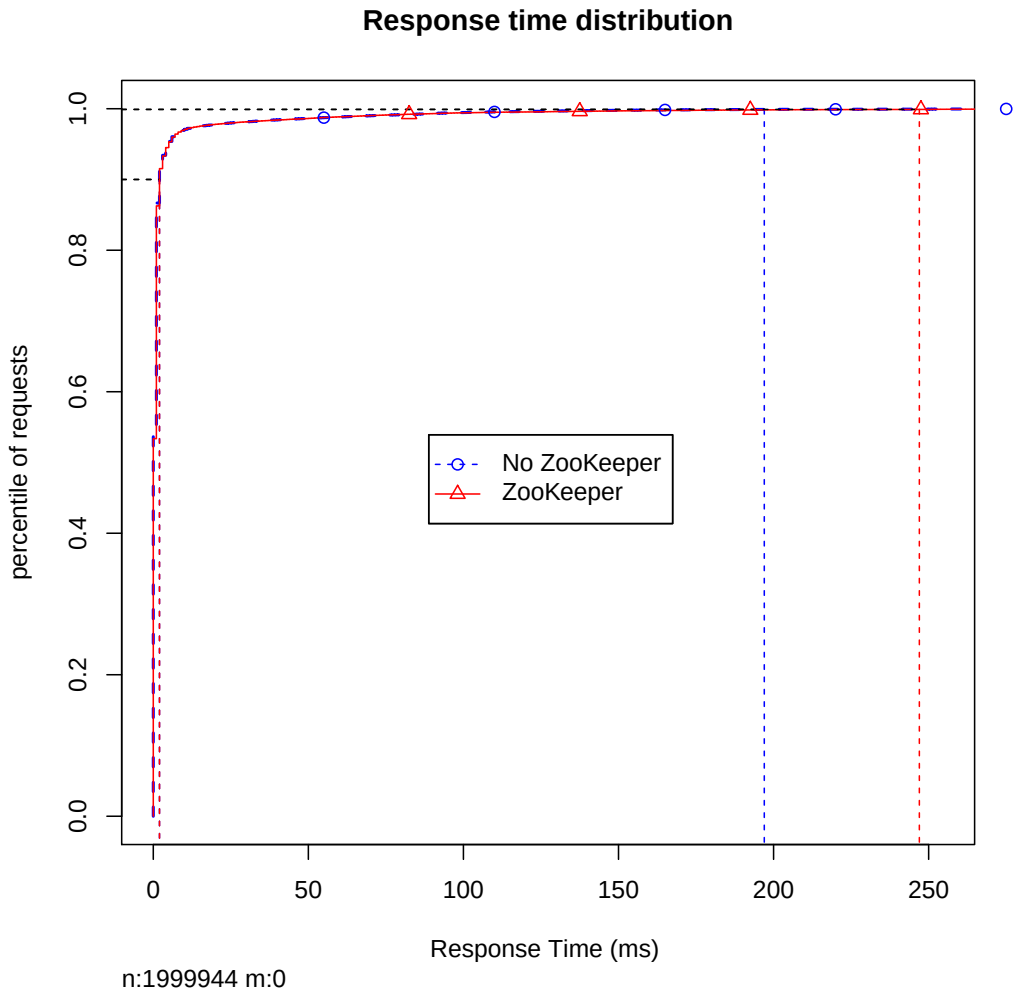


Figure 5.1: Response time distribution Core 2 Duo Mac mini

On Figure 5.1 we see that both implementations perform similarly on the Mac mini. The vertical lines mark the 0.9 and 0.999 percentile of requests. We see that 90% of the requests are serviced within 2 ms while at the .999

percentile the response time is at 197 ms for our implementation and 247 ms for the original code.

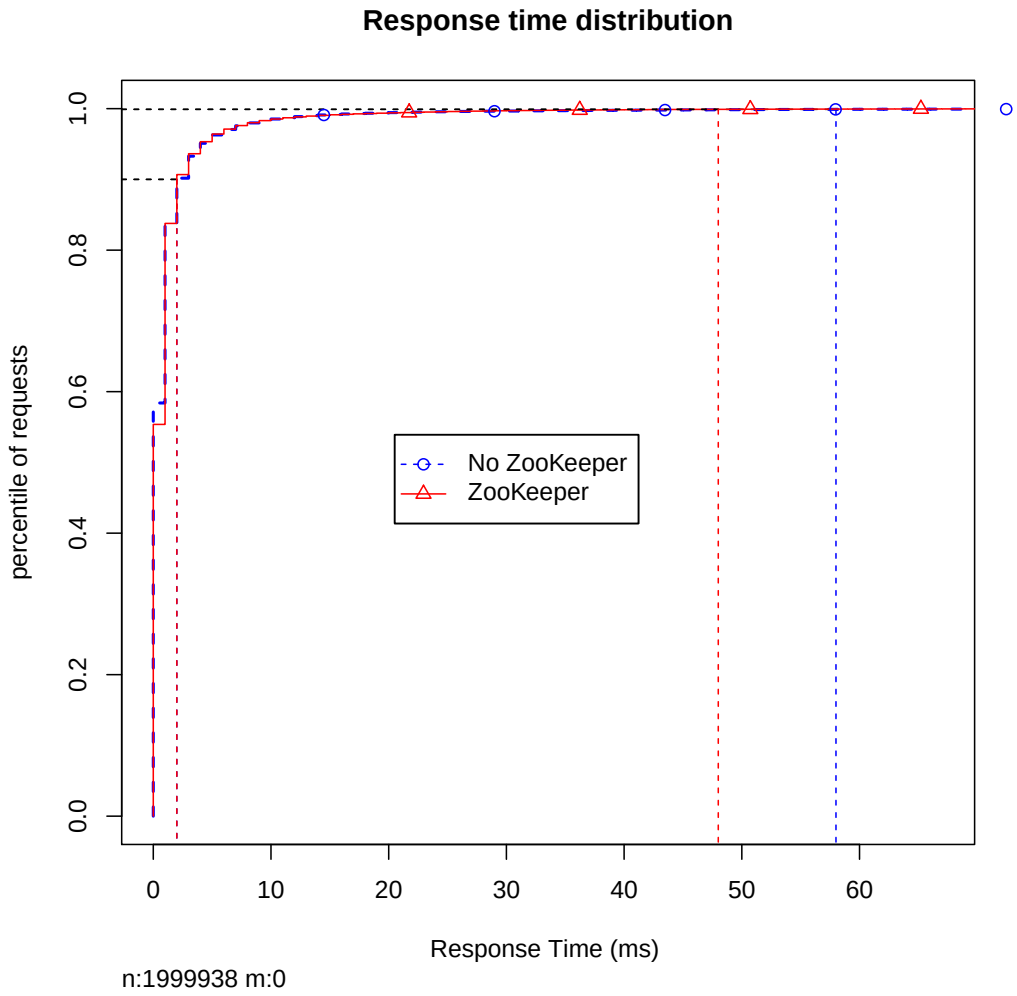


Figure 5.2: Response time distribution i5 MacBook Pro

On Figure 5.2 we see that the i5 MacBook Pro fares better. The vertical lines again mark the 0.9 and 0.999 percentile of requests. At the 0.90 percentile we see no improvement, however at the 0.999 percentile we see a noticeable improvement with 58 ms on our ZooKeeper implementation and 48 ms on the original code.

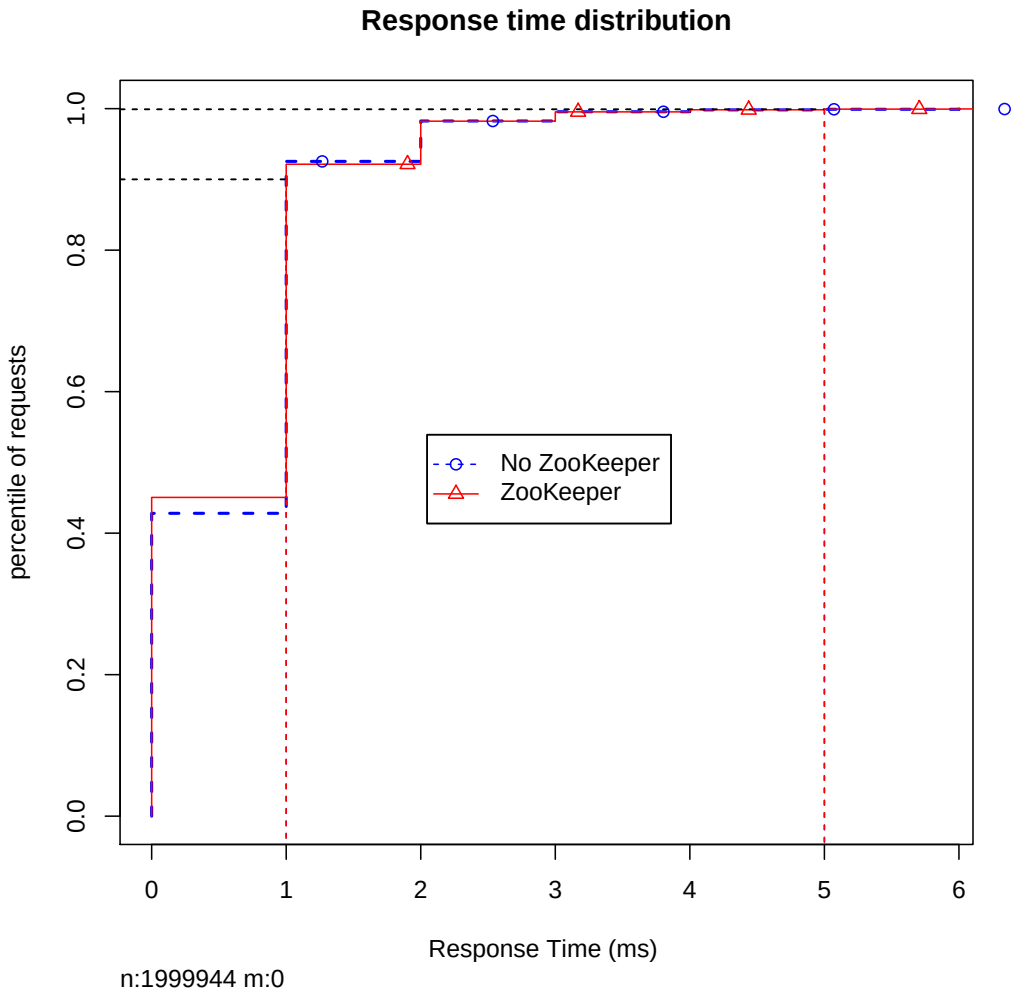


Figure 5.3: Response time distribution i7 MacBook Pro

Finally in Figure 5.3 we see the results on the i7 Macbook Pro. At the 0.90 percentile the system responds in 1 ms in both cases and again we see a great increase in performance at the 0.999 percentile with 5 ms for both systems.

5.3.2 Throughput

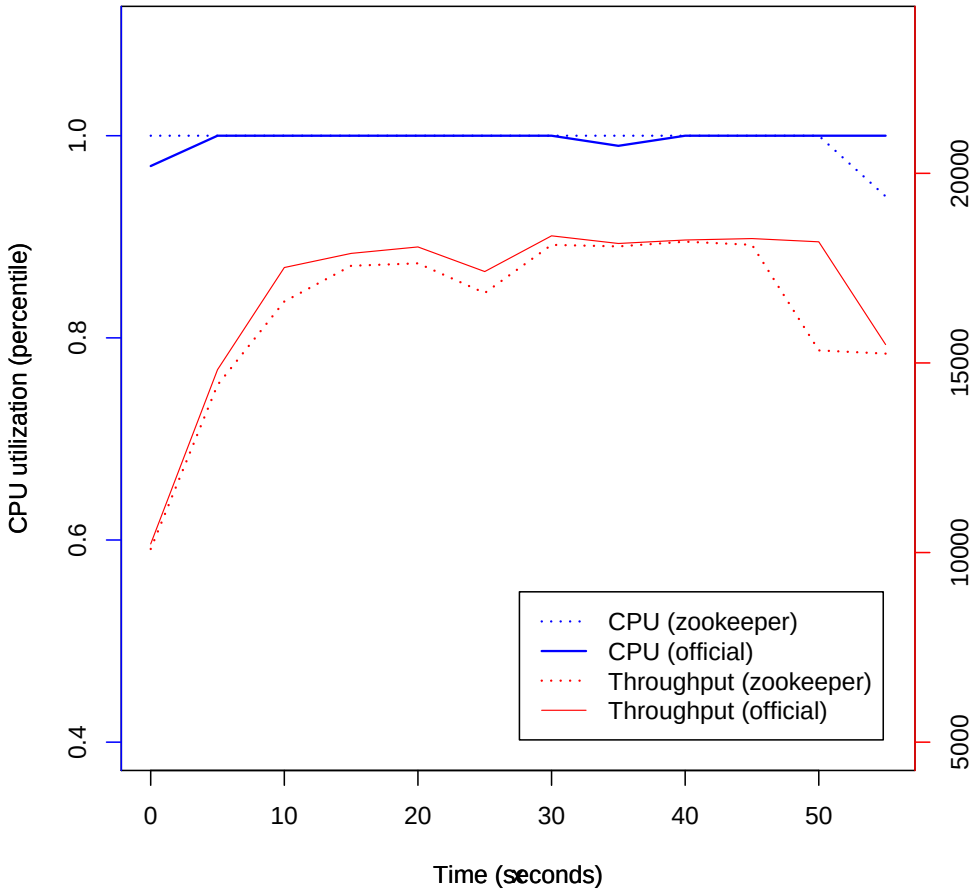


Figure 5.4: Throughput and CPU-load under stress test on Mac Mini

On Figure 5.4 we see that the Mac Mini's Core 2 Duo is clearly running at maximum capacity keeping up with the workload, running at close to 100% utilization of the CPU. Maximum throughput is hovering around 17k requests per second.

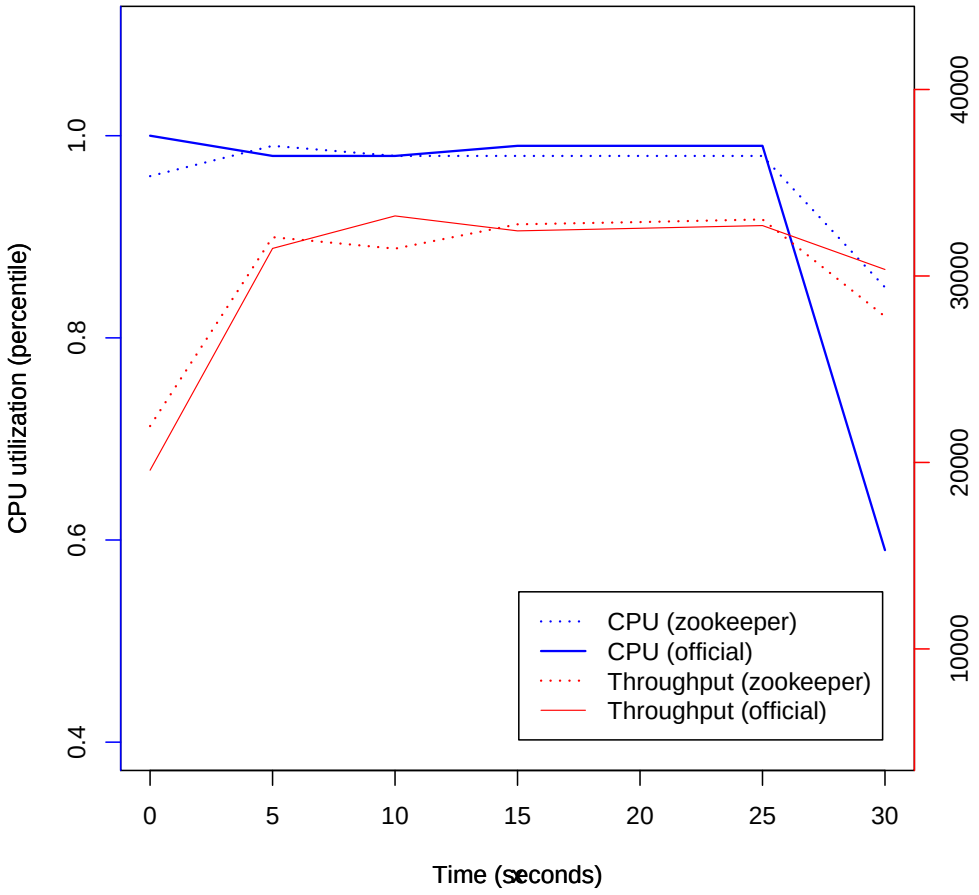


Figure 5.5: Throughput and CPU-load under stress test on i5 MacBook Pro

Figure 5.5 paints a very similar picture. The Core i5 in the MacBook performs a lot better than the Mac Mini, but it is still not able to keep up with the workload. It reaches saturation at around 32k requests per second.

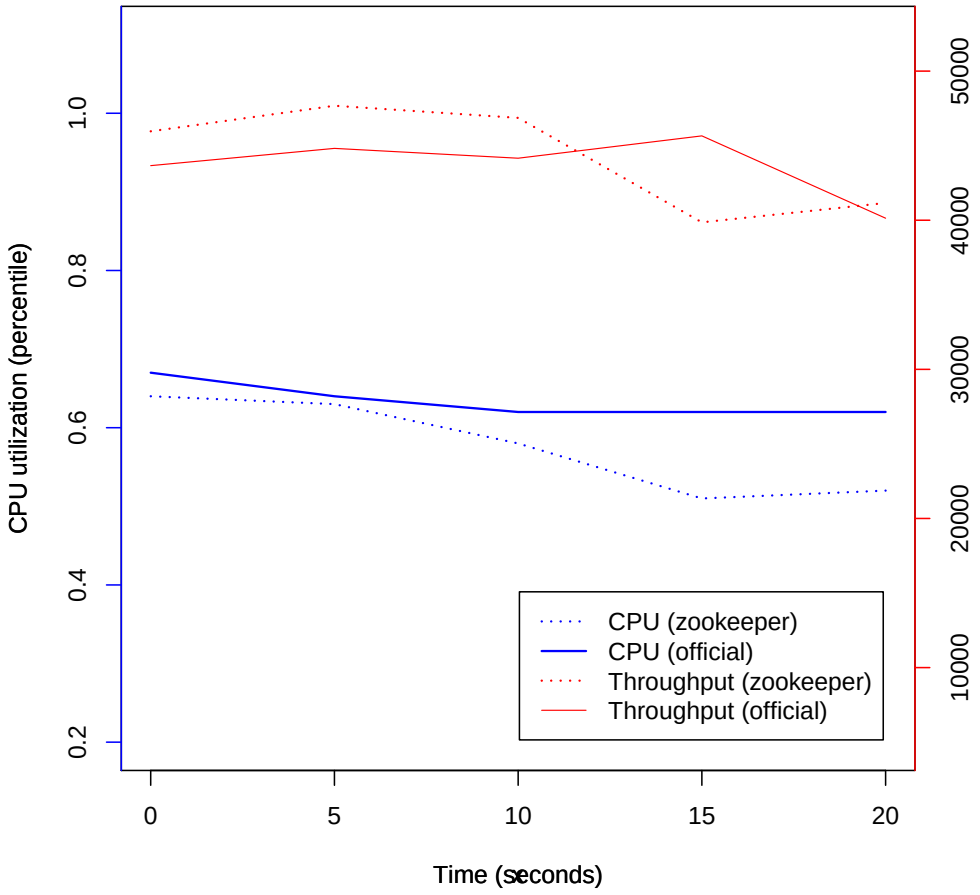


Figure 5.6: Throughput and CPU-load under stress test on i7 MacBook Pro

Finally we have the Core i7 Macbook. Figure 5.6 shows us the Core i7 saturated at around 44k requests per second. It is worth noting that in this experiment the CPU-utilization is at less than 80% meaning there is another factor limiting the system. Our explanation is that disk access might act as a bottleneck and thus limiting the i7 at 44k requests per second.

5.3.3 Adaptive cluster behavior

In this section follows the results form our adaptive cluster experiments.

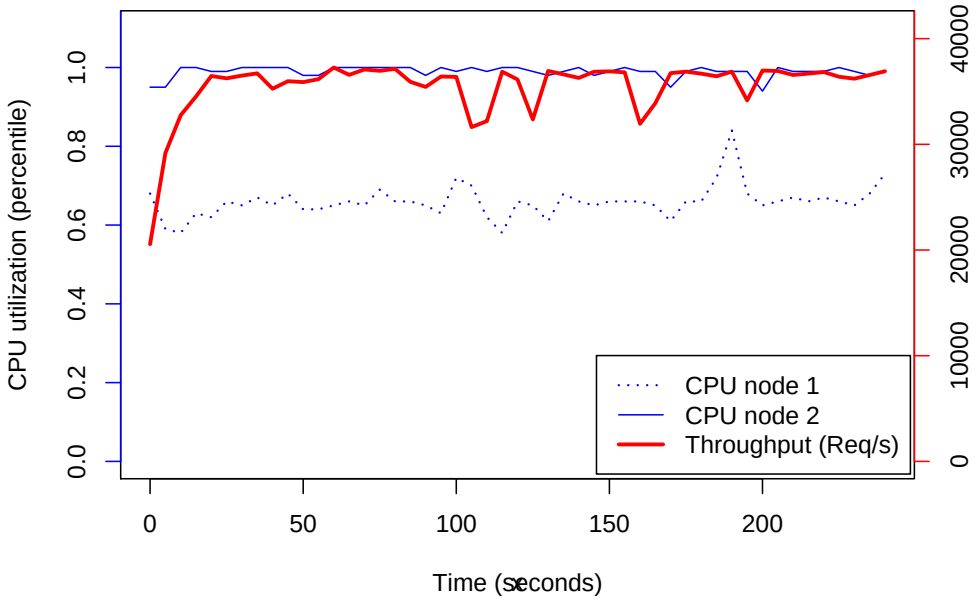


Figure 5.7: A baseline benchmark for a cluster of 2 nodes serving queries.

In Figure 5.7 we see a cluster of 2 nodes serving queries and their CPU utilization. The average throughput for these two is 36k requests/s, yet their individual performance is at 17k and 33k, respectively. We can also note that the CPU of node 1 is bound around 64%, where as node 2 is at above 95% the entire run.

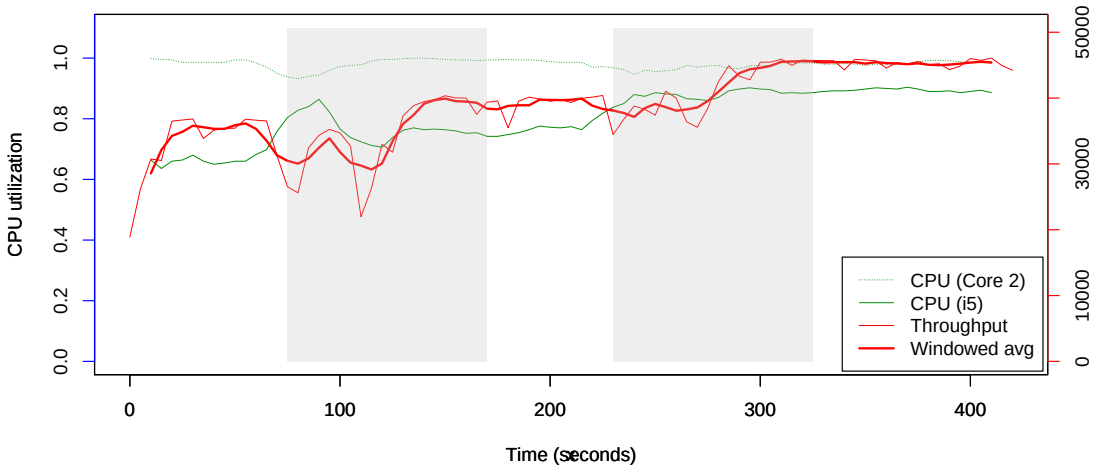


Figure 5.8: Manual moving of 2 partitions from a struggling node (Core 2). Grey regions mark rebalance period, where data is prepared and transferred.

In Figure 5.8 we see 2 manual rebalances. We see total CPU-utilization increase as we move partitions away from a struggling node and over to one less worked. The aim of this experiment was to get a baseline to compare our automatic rebalance service to. We see throughput steadily increase after each rebalance moving from around 36k up to 45k. During the rebalance we have two very distict drops in throughput. The first one is from when request starts being proxied and all client threads must reconnect. The second one is from the actual data transfer.

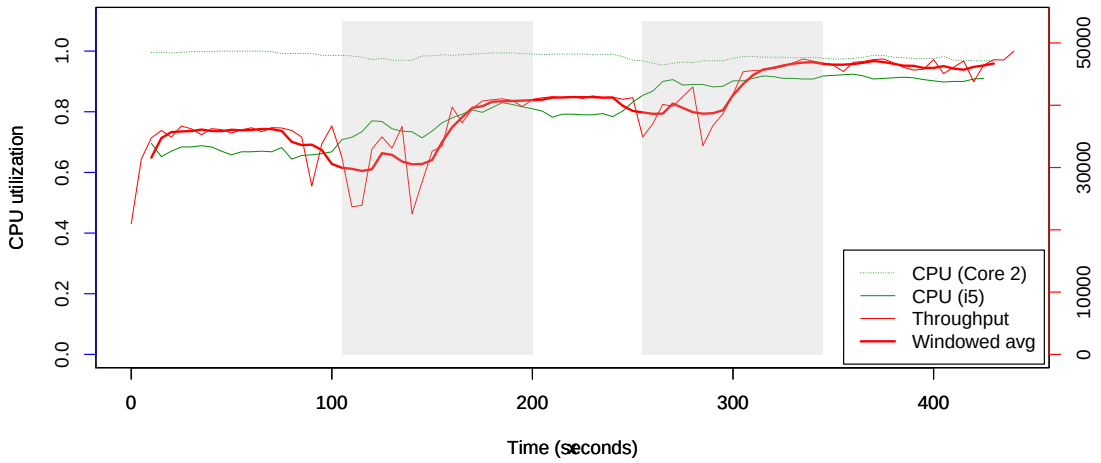


Figure 5.9: Automatic moving of 2 partitions from a struggling node (Core 2). Grey regions mark rebalance period, where data is prepared and transferred. CPU threshold set at `0.84`.

Figure 5.9 shows the same experiment just automated by Headmaster. We again see a steady increase in throughput as partitions are moved. We start out at about 36k requests per second, moving up to 46k requests/s as the load balances out. Also here we see two dips in performance per rebalance period. The first one stems from the clients reconnect after new metadata is discovered. The second one is during the moving of records between nodes.

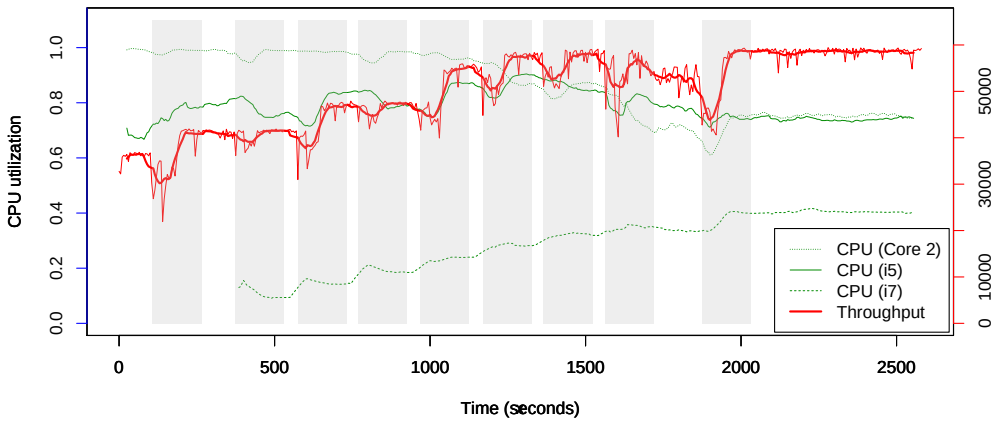


Figure 5.10: Automatic migration of partitions in a struggling cluster after a third node joins. A smoothed average is plotted on top of throughput, but is excluded from the legend for spatial reasons.

Figure 5.10 shows the result adding a node to a struggling cluster and rebalance live. At t=380 the i7 is introduced to the cluster. We see a steady increase in performance as more and more partitions are moved to the i7. In the end the different nodes are holding different number of partitions. This is 3,5,8 for the Core 2, i5 and i7 respectively.

At t=1700 there is a significant loss of throughput after a rebalance. We found no reason for this in our logs, so we suspect this to be related to unrelated work done by the operating system on the workload generating computer, but we failed to find any verification on this. The test is quite long, running for over 40 minutes and a quite a few of our tests got affected by random events on this time frame.

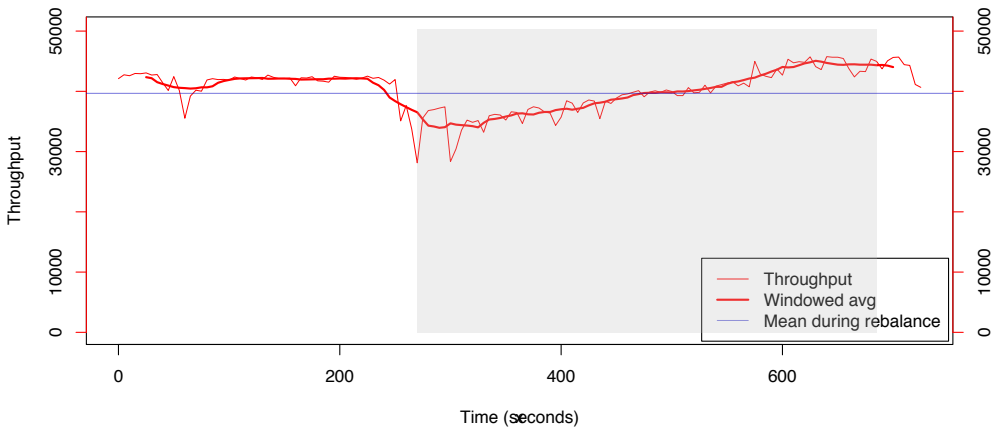


Figure 5.11: In depth view of thoughput during a big rebalance operation. The blue vertical line marks average throughput during the rebalance.

Figure 5.11 shows a large partition being moved. The partition holds almost 1 million entries or almost 1 gigabyte of data. Before the rebalance we have a steady throughput of 42k requests per second. For the duration of the rebalance, we average around 39,5k requests per second. We move a around 6% of all values and the rebalance causes a 6% loss of throughput, running at roughly 94% of performance before the rebalance.

Contrary to our expectations, we have a sudden drop in throughput followed by a steady increase during the rebalance. The sudden drop is caused by all clients being forced to fetch the newest metadata. The steady increase in throughput is a result of an increasing number of the entries being available at the stealer node, and lessening the need for proxying GET requests.

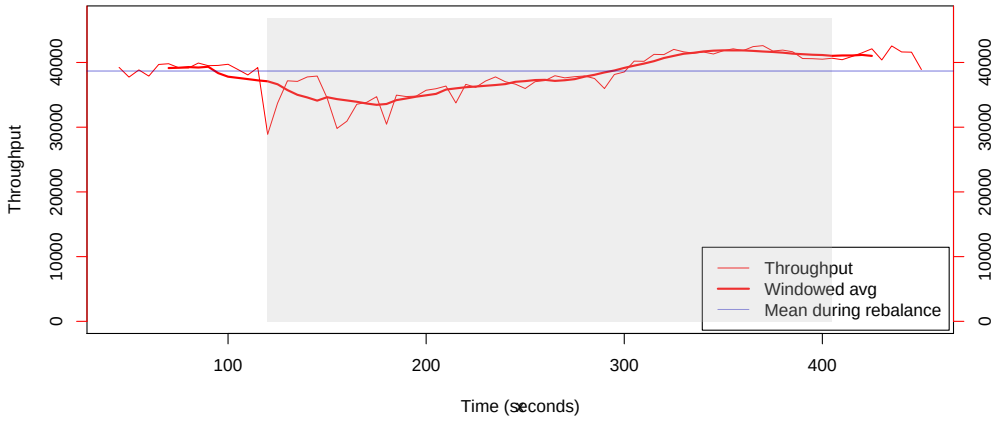


Figure 5.12: In depth view of thoughput during a rebalance with 15 million entries and 32 partitions. The blue vertical line marks average throughput during the rebalance.

In Figure 5.12 we see the results of moving one of the 32 partitions. Before the rebalance we service 39,2k requests per second, while during the rebalance we average 38,6k requests per second. This shows that over 98% of the cluster performance is maintained during the process. For comparison we now move 3% of all values.

Chapter 6

Evaluation

This chapter will cover discussion regarding our results. In section 6.1 we discuss our results from the experiments. Section 6.2 covers our thoughts on the system we have created as a whole and in section 6.3 we share some general experiences we had during the project.

6.1 Results discussion

In this section we will discuss our results as well as issues related to network limits.

6.1.1 Network limits

We did not have access to different network hardware while doing this project. Our computers proved very capable, with each individual node pushing 34-70 MB per second in raw network data. This made us quickly reach the network speed limit of our gigabit network infrastructure. During benchmarking the load generator was receiving over 130MB/s from the cluster. This limit was reached at around 60k requests per second each of 1024 kB values, providing a hard limit for our throughput. This is why throughput is capped at 60k in our experiments, even though there still are nodes with lots of idle CPU-cycles. On a 100Mbit connection we could only send or receive about 5k requets per second equaling to around 12 MB/s

6.1.2 Response time distribution and throughput

In our initial benchmarking experiments we have found no significant difference in the performance when comparing our ZooKeeper implementation

with the original Voldemort code. In some cases our implementation is performing slightly better, while in other the opposite is true. We interpret this shifting variance between them to suggest the two versions are quite similar. If the jitter between tests are seemingly larger than the impact of the changes, it is hard to find anything more conclusive.

Contrary to our expectations, the response time distribution varies greatly on different hardware. In our results the 0.999 percentile response time varies from 250 ms on the slowest hardware to 5 ms on the fastest. On the .90 percentile there is however is much less difference with response times of 2 ms on the slowest hardware and 1 ms on the fastest.

Single-node throughput results are more in line with what we expected. The slow Core 2 Duo is barely able to serve 17k requests per second while the Core i5 and i7 are able to serve 32k and 44k.

6.1.3 Scaling and balancing

To achieve full throughput from scaling, some skewing of partitions is required. In our adaptive cluster experiments we have compared automated to manual rebalancing and found our automatic solution to perform on par with the manual one. We have also done a extensive cluster expansion test and monitored CPU-utilization and throughput. These experiments show that our implementation works as intended.

With optimal partition distribution, a cluster consisting of the two slowest machines was able to service 46k requests per second providing almost the sum of both individual nodes. We also see that a cluster of all three nodes with proper partition distribution easily was able to service the maximum request rate of 60k requests per second.

6.1.4 Cost of rebalance

The last two results focuses on the performance cost of transferring partitions live. We see how partition-size affects performance loss during a rebalance. Our results suggests that performance cost is tied to the percentile of keys affected by the rebalance. With 98% performance retained when 3% of keys are copied, and 94% retained performance when 6% of keys are copied we would suggest taking this into consideration when deciding number of partitions.

Contrary to our expectations, we have a sudden drop in throughput followed by a steady increase during the rebalance. The sudden drop is caused by all clients being forced to fetch the newest metadata. The steady increase in throughput is a result of an increasing number of the entries being available

at the stealer node, and decreasing the need for proxying GET requests to the donor.

We would also argue this makes sense in the relationship between keys moved and performance loss. When the proxy bridges are setup, all requests for the affected keys are routed to the stealer node, which do not yet hold any of these keys. This means the percentage of keys being moved, are now temporarily *unavailable*, in the sense that all requests for this percentile of keys must be proxied to the donor node and fetched, before the query can be answered. This performance cost should in other words relate to the percentage of keys affected.

6.1.5 Issues

Before starting we were worried of getting into scenario where two or more nodes ends up endlessly juggling partitions. We deemed this the biggest weakness of such a automated system. This situation did however never occur in our testing. We still think this is something that potential users need to be aware of, and need to watch out for. To mitigate this behavior, having a high number of partitions to minimize performance impact of moving one, seem like a good option.

Having a high number of partitions have been a point we have made several times in this thesis. There is however a draw back. If there is a very high number of partitions in the system, and not a lot of data, we could have situations were moving a partition has little to no impact on the target systems. This would lead to waste of network and system resources for little to no gain. This could be somewhat mitigated by moving several partitions at once, but we consider this to be more error prone.

6.1.6 Other notes

Overall we found the performance to be very consistent. Our results were on large very reproducible and saw little variance between benchmark runs in the longer tests. The shorter benchmarks of the individual nodes did show some variance between versions, but we could not find any consistent patterns.

6.2 System evaluation

Overall we are happy with our results. They show that we have not noticeably affected the performance of Voldemort when adding automatic scaling. For

now the monitor service is started as a separate process on each node involved in Voldemort. This could be incorporated into Voldemort, but we think Headmaster should be isolated for stability reasons. Crashing Headmaster should not affect your database.

For now Headmaster is implemented as a standalone service so that it can be run remotely, and independently from Voldemort. While working, we had at one point inadvertently started two Headmasters at once, which kept fighting for control. We spent a fair amount of time debugging weird race conditions until we realized there was two services operating at the same time. It is now a master based service with leader election through ZooKeeper that can run on any number of nodes, without fighting for control. The only dependency for any operation or function is an available ZooKeeper instance. The cluster still works fine even if Headmaster is down, not working or crashed.

Our system is best used in diverse clusters of heterogeneous nodes where Headmaster can help automatically distribute the partition set across different nodes to improve cluster load balance. As the performance loss while rebalancing is not too bad, it could be possible to utilize Headmasters rebalancing before peak hours to alleviate struggling nodes. This of course is heavily dependent on partition size. A large amount of partitions will give smaller amounts of data to move, but if each partition is only held by a single node, nothing can be done even if you have many nodes and partitions. They are just too large to move in this scenario. It is therefore of significant importance to assign enough partitions for a huge data set when first creating the cluster.

Our work should be rather failure tolerant, but we have not done very extensive failure scenario testing, there might still be some bugs.

6.3 General experiences

In this section we cover general experiences we have made and problems we encountered that we find worth sharing.

Server side caching

While testing, we noticed no real difference in performance between a 12MB and 2GB database cache on our Voldemort servers. This might suggest there is a problem with the cache setting in Voldemort. All of our hardware run on SSDs, but we would still consider this result surprising. This could be investigated further using largely different of data sets, but this was not our

main focus in this thesis. However, a theory is that the data files fit into the OS file caching, still providing very good read speeds.

Client updates

While benchmarking rebalancing we found a critical bug we had introduced in the way clients receive updates of configuration. It caused values being moved during a rebalance to be unavailable for the entire rebalance duration. This would have been hard to discover if we did not compare with the original Voldemort implementation. A graph illustrating the issue is available in figure 6.1.

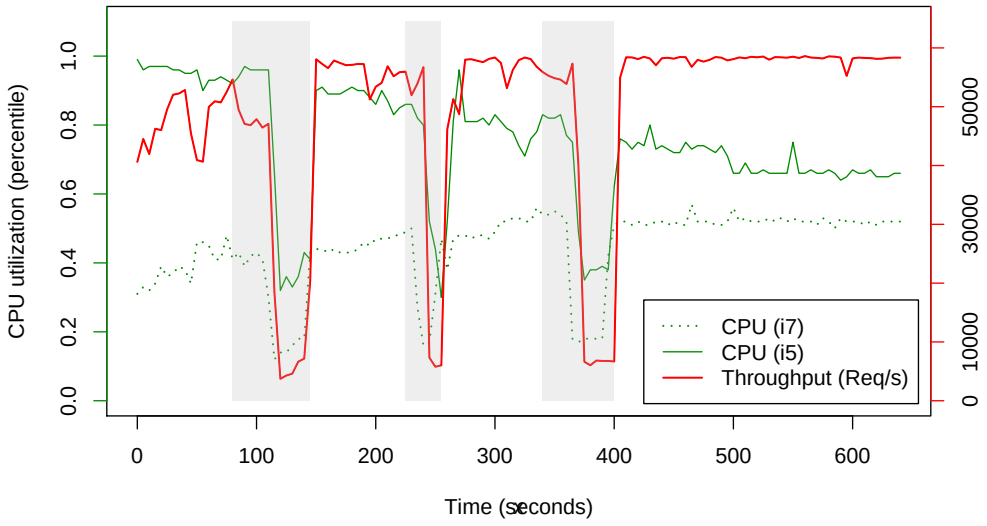


Figure 6.1: Fatal bug causing data moved during a rebalance to be inaccessable for close to the entire duration of the rebalance, causing major drops in throughput.

VectorClock of Stores

We also had issues with translating a config files version number in ZooKeeper to a valid vector clock in Voldemort. When a config file is committed into Voldemort internally, it is supposed to follow the same rules for vector clock versioning as normal keys. The bug caused the rebalance cluster.xml and stores.xml sent to nodes when getting ready for a rebalance to be rejected,

failing the rebalance attempt. The bug was also only discovered after the first rebalance in a single session, so it took some time until we encountered it.

It is related to how the StoreDefinition object is versioned internally and was quite hard to track down. When the first rebalance occurs, the new StoreDefinition is saved to the *cache* only, with a incremented version clock. When the next rebalance occurs, the StoreDefinition is fetched from the persistent store. This version does not include the vector clock from the previous one stored in the cache, so when the new StoreDefinition is saved, the vector clock has the same or an old value and is rejected by the store.

We circumvented this by making sure we fetch StoreDefinitions from the cache only when we are rebalancing, preserving the vector clock between runs.

Chapter 7

Summary and conclusion

7.1 Summary

Through our project we have fundamentally changed how Voldemort stores configuration data and utilized ZooKeeper to provide powerful new features. We have created Headmaster to administrate the cluster and allow for new nodes to automatically be included in the cluster. Finally we have created a monitor service using the SigarAPI and the decision taker StatusAnalyzer. Through our experiments we have verified that our implementation does not suffer any noticeable performance loss and successfully migrates partitions to achieve a more balanced cluster, and more optimized workload distribution.

7.2 Conclusion

Management of a cluster consisting of lots of nodes is cumbersome and surprisingly error prone. We have shown that a more automatic approach can be viable, especially in terms of delivered performance. Automating management and scaling of such systems can be immensely helpful, but there is still work to be done before it can manage a live production system.

Management of distributed systems becomes increasingly difficult as the number of nodes increases. Scaling a system to meet computing requirements is also not a trivial task, because demands change over time. We have successfully created a management system for Voldemort that automates some of these tasks, automatically adjusting the system according to its current load and assists in maximizing throughput on available resources.

Chapter 8

Further work

We will here introduce and expand upon some ideas we have for further work on this topic. Some ideas we did not find the time for. Others are not as of this writing possible yet.

8.1 Downscaling

Voldemort as of this project, does not support downscaling, ie. downsizing the cluster. There is work going on for this, but it is not a current priority. We would of course like to include this functionality into our project once it is possible.

8.2 Data rebalancing

A cluster serving requests might over time develop an uneven data distribution like in Table 8.1. The reason for this uneven data distribution is simply bad luck.

If certain partitions end up with a skewed amount of data, even shuffling partitions around to idle nodes might be fruitless. One might have to do a very expensive rehash, effectively moving the partition key space. If this could be done live in an efficient manner, it could greatly improve the balancing and performance of each node in such edge cases.

Node	Partition	data %
0	0	12.5
	1	12.5
1	2	12.5
	3	24.0
2	4	1.0
	5	5.0
3	6	20.0
	7	12.5

Table 8.1: A cluster where partitions exhibit uneven data distribution. This cluster *potentially* needs rebalancing of data.

8.3 Ondemand computing

With the recent years advancement in cloud computing, especially those driven by Amazons EC2 service, it is now easy to rent computing capacity as you go and pay for what you use. With the current design, having physical computers ready is an absolute requirement, ie. to insert a new node one needs actual hardware.

An idea we have for future work is a module to the Headmaster rebalancer that can automatically create, setup and assign Amazon EC2 machines on demand. This would allow for scaling the cluster up and down on demand with rented computing.

8.4 Decision support system

We think it would be interesting with further decision support for scaling. We currently don't do anything with regards to available disk space or memory usage. Combining this with the possibilities in ondemand computing would be interesting. This could allow for ordering and extending both disk space and memory in running nodes, which is currently possible with e.g. Amazon Web Services.

8.5 Running with backup nodes

One could assign a certain znode path or a certain property with new nodes in `/active` to add a node as a backup node. A backup node can be a queued node that is kept ready for use in case another node dies permanently. E.g. if a node has a permanent failure, this backup node is put in to replace

it immediately. A challenge here could be false positives, replacing a node before we need to.

One could also see backup nodes waiting on znodes in `/active` to disappear, then immediately replace them. This does not seem like very efficient use of resources though.

8.6 Real world testing

As we have done this project without a real application using Voldemort, we do not have access to realistic test data. We feel a few more real world tests should be done before our management system can be deployed to a live production environment.

It would also be interesting to apply it to a already running instance and see if and what improvements could be made, both to the code and to the running cluster. Seeing how and if a running real world cluster can gain anything from the automatic load adaption would help furthering this kind of automatic management.

8.7 Management interface

Another idea we got while working on this project, is live monitoring of status in a web based interface. Managing nodes in the cluster through a web interface could add a lot of value, especially for system administrators. Manually configuring and monitoring large clusters of over 100 computers is tedious and error prone, if not impossible by humans. Using our framework to automate such tasks through a interface would not be too hard to implement. We can easily see it being used for both introducing new nodes, replacing and overall monitoring of system health.

Bibliography

[1] Apache zookeeper: A distributed cordination service. https://zookeeper.apache.org/. [Online; accessed 24.3.2014].

[2] Automatic scaling of cassandra clusters. https://www.duo.uio.no/bitstream/handle/10852/36663/Baakind-Master.pdf?sequence=1. [Online: accessed 22.05.2014].

[3] Our voldemort implementation. https://github.com/esiqveland/voldemort/tree/zookeeper. [Online git repository].

[4] This thesis report code. https://github.com/esiqveland/masteroppgave. [Online git repository].

[5] Understanding hbase. http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable. [Online; accessed 20.3.2014].

[6] Voldemort: A distributed database. http://www.project-voldemort.com/voldemort/. [Online; accessed 14.1.2014].

[7] Why tellybug moved from cassandra to amazon dynamo. http://attentionshard.wordpress.com/2013/09/30/why-tellybug-moved-from-cassandra-to-amazon-dynamodb/. [Online; accessed 29.04.2014].

[8] Yahoo cloud serving benchmark. http://research.yahoo.com/Web_Information_Management/YCSB/. [Online: accessed 20.03.2014].

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *IN PROC. SOSP*, pages 205–220, 2007.

[10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In

Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[11] Frank E Harrell Jr, with contributions from Charles Dupont, and many others. *Hmisc: Harrell Miscellaneous*, 2014. R package version 3.14-4.

[12] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy Abstract. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In Proc. 29th ACM Symposium on Theory of Computing (STOC*, pages 654–663, 1997.

[13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[14] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, August 2012.

[15] VMWARE. Hyperic sigar api. [Online: March 2014] http://www.hyperic.com/products/sigar.