Repartition optimization for distributed B+ trees

Master's Thesis

May 1, 2019

Paolo Aurecchia
paolo.aurecchia@usi.ch

supervised by
Paulo Coelho

Master of Science in Financial Technology and Computing
Department of Computer Science
Università della Svizzera italiana

Abstract

Contents

1

Introduction

1.1 Motivation

1.2 Previous Work

1.3 Main Objective

1.4 The Structure of this Report

2

Background

In this section we will go over some of the fundamentals of distributed systems and algorithms that will help to better understand the later parts of this report.

We will first give a brief introduction to the main challenges of working in distributed environments. Then we will discuss a fundamental problem of distributed systems known as Consensus, followed by the description of one of the most popular algorithms used to solve Consensus in distributed settings, called Paxos. Lastly we will introduce GeoPaxos, an improvement over Paxos which aims to provide better performance in geo-distributed environments.

2.1 Distributed Systems

Distributed systems can be encountered in a multitude of situations in our everyday lives, and their importance is growing every year: Entertainment services such as music and video streaming services, banks, hospitals, social medias, web engines are just a few of the many types of distributed systems that we use on a daily basis, and each of these may have different challenges and needs when it comes to handling distribution.

Even a clear definition of a distributed system is hard to find. In general, we can describe it as a collection of indipendent machines that communicate and coordinate their work to provide a certain service to its users: to them, the system should appear as a single and homogeneous entity, hiding its distributed nature.

Working with distributed environments has some inherent challenges, some more important than others depending on the specific system. Concurrency issues, for example when multiple users want to withdraw money from a bank account. Ordering and synchronization, because the different machines will most likely work with different clocks and their communication may take a noticeable amount of time. Failure resiliance, so that the system does not go down when one of its machines fails. These are only a few of the many problems that can and will be encountered when working with distributed systems.

To tackle these problems, The choice of the models used to represent them is fundamental: choosing a model too strong may make the problem easier to solve, but it may represent an unrealistic view of the problem at hand. On other hand, weaker models may make the problem impossible to solve without further assumptions.

A synchronous system is one where we know that the process speed and the time delay for the communication are bounded; this helps us simplify our view of the timing and communication side, but it will usually not be a realistic representation of the real world system.

On the other hand, an asynchronous system is one that does not have bounds on process speed and time delay, which is a more general assumption which could greatly increase the difficulty of solving the problem.

Similarly, for a model we could have reliable channels, where if a process sends a message and the receiver does not crash then the receiver will get the message, or quasi-reliable, where both sender and receiver must not crash for the receiver to get the message.

It is also important to model the possible failure of the processes. Some types of failure models can be:

- **Failstop** the process fails by halting execution, and it does not recover. Other processes will know about the failure of the process
- **Crash** the process fails by halting execution, and it does not recover. Furthermore, the other processes may not be able to identify its failure
- **Byzantine failures** the process fails by exhibiting arbitrary behavior, which could be unexpected replies or malicious actions

Again, the choice of the failure model gives us different degrees of difficulty: a failstop model makes the problem easier to solve, but a system that is able to tolerate byzantine failures is going to be much more resiliant to failures.

2.2 Consensus

The problem of consensus is a fundamental problem in distributed systems: it was first introduced in the early 80s [**?**], [**?**] and since then it remained important in most distributed environment, ranging from more classical server applications to the more recent Blockchains.

Consensus is basically an agreement problem: the goal is to have a set of entities reach an agreement on a value that was previously proposed by (at least) one of those entitities. This definition is quite general, and for a good reason: depending on the different assumptions we make on the system model, there are stronger and weaker types of consensus that can be achieved.

The three core properties of a consensus protocol are the following:

- **Termination** Eventually every correct process decides on one value
- **Uniform integrity** If a process decides on a value, then this value was proposed by some process
- **Uniform agreement** No two processes decide on a different value

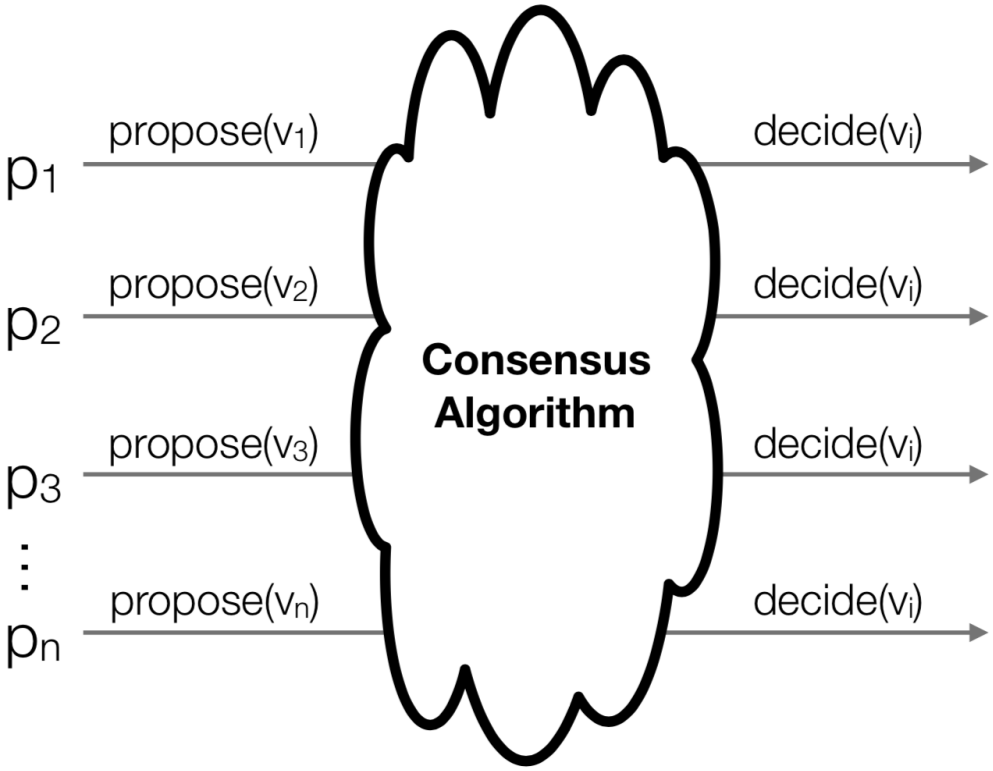

Figure 2.1: The architecture of the system. The *Viper IDE* and *Viper Debugger* boxes denote, respectively, the main Viper extension and the new debugger extension. They both interact, independently, with Visual Studio Code. Viper Server is responsible for running the verification backends.

In this section we will go over some of the more relevant types of consensus, and respective implementations that are able to achieve consensus in the given system models.

2.2.1 Synchronous model

In a synchronous system model, consensus is quite straightforward to achieve. Let us assume to have a set of n processes, that will be the entities that want to reach a common agreement upon a proposed value. Let us also introduce the primitives used by the

processes: each process is able to send and receive messages, through the primitives $send(m)$ and $receive(m)$ respectively, to communicate their intentions to the others. The failure model in a crash failure model, hence a correct process will never crash, while a faulty one eventually will; the number of faulty processes f.

Since we are in a synchronous model, we have a bound on the maximum time delay of a message, which means that a process is sure that if a message is sent, it will be received by the receiver before the maximum time delay. This is a quite strong and unrealistic model, but it makes the algorithm to solve consensus quite simple.

Let us have the algorithm proceed in rounds, where each round is as follows:

1. Each process sends to all other processes a message containing information about its state.

2. At the end of the round, after the maximum time bound, processes will have received a set of messages from a subset of all processes.

3. The processes will then use the received messages together with its current state to reach a new state.

The state can be simply a set values received until that moment, with the initial set containing only our own proposed value. Also, in a round a process can receive only a subset of messages from the other processes, since we can have up to f faulty processes.

After $f + 1$ rounds, we can be certain that each correct process will have enough information to deterministically decide on a single proposed value, and therefore reach consensus. This is because if we have a round where no process fails, then every process will receive all other values, and therefore we will be able to decide on a value. Also, since we have at most f faulty processes, we can have at most f rounds with a failing process. Therefore, we can be certain that $f + 1$ rounds will be enough to reach agreement.

[should I put the lemmas properly?]

2.2.2 Asynchronous model

In this model, we have no bounds on process speed and maximum message delays; this means that a process if or when a message will be received by the other processes. This is a much weaker model compared to the previous one, and as a matter of fact it is proven that consensus cannot be solved in such a model [FLP proof].

2.2.3 Alternative models

The synchronous model allows us to solve consensus, but it's too strong of a model to be of any practical use. The asynchronous model cannot solve consensus. Therefore, we either have to find a system model that is weaker than synchronous and stronger than asynchronous, or we have to weaken the problem.

An alternative system model would be a partially-synchronous one, where the messages are asynchronous up to a GST (Global Stabilization Time), after which we can assume communication to become synchronous.

Another option is to still use an asynchronous system, but to also elect a process to be the leader and guide the whole decision procedure; an example of such an algorithm is Paxos, which is presented in the next section.

We could have failure detectors (which can be used with varying levels of completeness and accuracy), which could allow us to figure out when a process has failed, and therefore stoping us from waiting indefinitely for a message like in the basic asynchronous model.

An example of weaker problem definition would be to allow, for example, to have multiple values decided.

2.3 Paxos

Paxos is an algorithm to achieve consensus that relies on an asynchronous system model, which makes it particularly interesting for real world application where we don't have completely reliable communication channels. To achieve this, the Paxos algorithm needs to elect one process as a leader that will act as a coordinator during the various phases of the protocol.

In Paxos, there are four types of process roles:

- **Proposers** the processes that want to propose a value to be decided
- **Acceptors** the processes that take part in the voting part of the protocol; a quorum of acceptors is needed
- **Learners** the processes that, at the end of the algorithm, will be notified about the value decided by the algorithm
- **Leader** the process that acts as a coordinator

The election of the leader can be achieved in many different ways, as long as there is a point after which we have a unique correct process that is identified by the other processes as the leader.

The algorithm is divided into tree phases. During phase 1, a proposer that wants to propose a new value sends a new arbitrary value c-rnd higher than any previous c-rnd to the acceptors, so that all of them know that this is the most recent proposal attempt. Each acceptor, if the received c-rnd is truly the highest one it has received, replies to the proposer with its current state (rnd, v-rnd, v-val) so that the proposer know if there was already a previous proposed value, or if the proposer is allowed to propose its own.

In phase 2, once the proposer has received a reply from a quorum of acceptors, it checks if there was any acceptor that had previously voted for a value, and if this is the case, it picks the most recent one as the value to propose. If instead no acceptor had previously voted for a value, the proposer will be able to propose its own value. The proposer will then send to the acceptors the actual value to be proposed. The acceptors, if the message of the proposer is still the most recent on-going proposal, will send back a message as an acknowledgement back to the proposer.

In phase 3, once the proposer receives again a quorum of replies from the acceptors, will be allowed to send to the learners the newly decided value.

The pseudo-code for the phases is as follows:

```
  c-rnd : highest-numbered round the process has started
c-val : value the process has picked for round c-rnd
% rnd : highest-
numbered round the acceptor has participated, initially 0 v -
rnd : highest-
numbered round the acceptor has cast a vote, initially 0 v-
val : value voted by the acceptor in round v-rnd, initially null
% To propose value v:
% increase c-rnd to an arbitrary unique value send (PHASE 1A, c-
rnd) to acceptors
% upon receiving (PHASE 1A, c-rnd) from proposer if c-rnd > rnd then
% rnd ← c-rnd
% send (PHASE 1B, rnd, v-rnd, v-val) to proposer
% Phase 1B Phase 1A Acceptor Proposer
% upon receiving (PHASE 1B, rnd, v-rnd, v-val) from Qa such that c-
rnd = rnd k ← largest v-rnd value received
% V ← set of (v-rnd,v-val) received with v-rnd = k
% if k = 0 then let c-val be v
% else c-val ← the only v-val in V
% send (PHASE 2A, c-rnd, c-val) to acceptors
% upon receiving (PHASE 2A, c-rnd, c-val) from proposer if c-
rnd >= rnd then
% v-rnd ← c-rnd
% v-val ← c-val
% send (PHASE 2B, v-rnd, v-val) to proposer
```

```
% upon receiving (PHASE 2B, v-rnd, v-val) from Qa
% if for all received messages: v-rnd = c-rnd then
% send (DECISION, v-val) to learners
upon receiving (PHASE 1B, rnd, v-rnd, v-val) from Qa such that c-
rnd = rnd k ← largest v-rnd value received
V ← set of (v-rnd,v-val) received with v-rnd = k
if k = 0 then let c-val be v
else c-val ← the only v-val in V
send (PHASE 2A, c-rnd, c-val) to acceptors
upon receiving (PHASE 2A, c-rnd, c-val) from proposer if c-
rnd >= rnd then
v-rnd ← c-rnd
v-val ← c-val
send (PHASE 2B, v-rnd, v-val) to proposer
upon receiving (PHASE 2B, v-rnd, v-val) from Qa
if for all received messages: v-rnd = c-rnd then
send (DECISION, v-val) to learners
```

- **Termination** Eventually every correct process decides on one value
- **Uniform integrity** If a process decides on a value, then this value was proposed by some process
- **Uniform agreement** No two processes decide on a different value

[should I add correctness, liveness, and so on?]

Remember that this algorithm allows only for the decision of one value; once value is decided, another instance of Paxos has to be initiated. Hence, for realistic use cases one would use system that allow to run multiple instances of paxos in a row, such as MultiPaxos.

The presented version of Paxos is the simplest one. Liveness in this case is not guaranteed; to ensure it, only the leader should be allowed to propose values, since otherwise it is possible that two proposers keep stealing their turn from each others without making any progress. Therefore the proposers should forward their values to the leader that will then try to propose them.

Various other improvements can be performed: for example, the leader could start phase 1 before a value is proposed, which means that once a value is ready to be proposed we can start directly from phase 2, virtually removing the delay of phase 1. Also, at the end of Phase 2B the acceptors could send the decided value directly to the learners instead of having to first send it to the proposers; this allows to remove a whole message delay from the algorithm.

[should I write best message delays?]

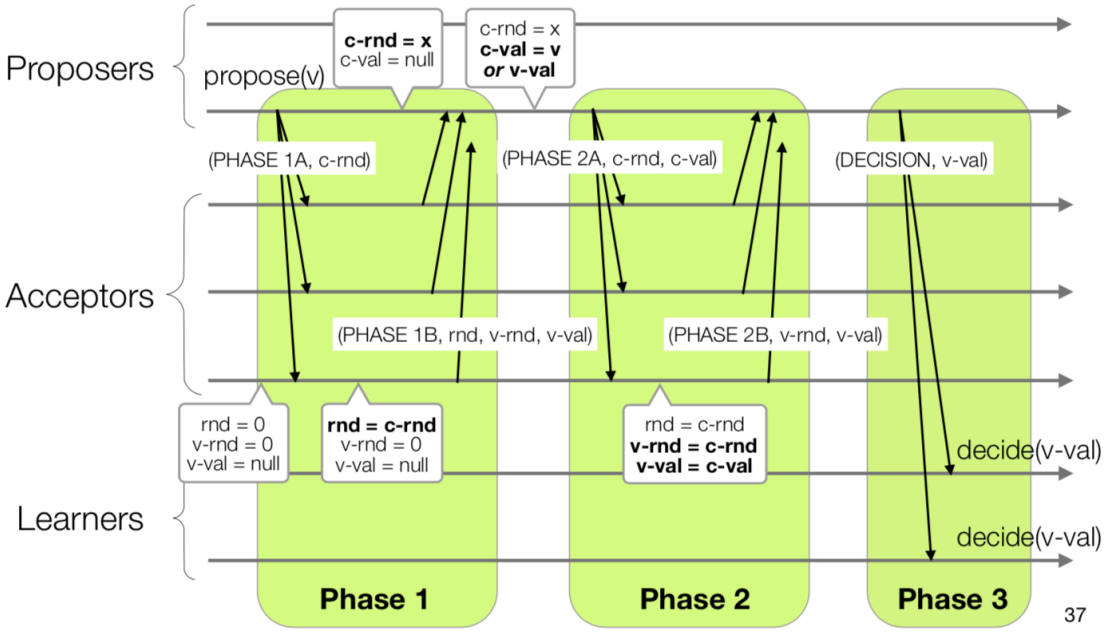

Figure 2.2: The architecture of the system. The *Viper IDE* and *Viper Debugger* boxes denote, respectively, the main Viper extension and the new debugger extension. They both interact, independently, with Visual Studio Code. Viper Server is responsible for running the verification backends.

2.4 GeoPaxos

Many current online services must serve clients distributed across geographic areas. In order to improve service avail- ability and performance, servers are typically replicated and deployed over geographically distributed sites (i.e., datacen- ters). By replicating the servers, the service can be configured to tolerate the crash of nodes within a single datacenter and the disruption of an entire datacenter. Geographic replication can improve performance by placing the data close to the clients, which reduces service latency.

Designing systems that coordinate geographically dis- tributed replicas is challenging. Some replicated systems resort to weak consistency to avoid the overhead of wide-area com- munication. Strong consistency provides more intuitive service behavior than weak consistency, at the cost of increased latency. Due to the importance of providing services that clients can intuitively understand, several approaches have been proposed to improve the performance of geo-distributed strongly consistent systems (e.g., [1], [2], [3], [4]). This paper presents GeoPaxos, a protocol that combines three insights to implement efficient state machine replication in geographically distributed environments.

First, GeoPaxos decouples ordering from execution [5]. Although Paxos introduces different roles for the ordering and execution of operations (acceptors and learners, respectively [6]), Paxos-based systems typically combine the two roles in a replica (e.g., [1], [2], [7]). Coupling order and execution in a geographically distributed setting, however, leads to a performance dilemma. On the one hand, replicas must be deployed near clients to reduce latency (e.g., clients can quickly read from a nearby replica). On the other hand, distributing replicas across geographic areas to serve remote clients slows down ordering, since replicas must coordinate to order operations. By decoupling order from execution, GeoPaxos can quickly order operations using servers in dif- ferent datacenters within the same region [1] and deploy geo- graphically distributed replicas without penalizing the ordering of operations.

Second, instead of totally ordering operations before execut- ing them, as traditionally done in state machine replication [8], GeoPaxos partially orders operations. It is well-known that state machine replication does not need a total order of operations [9] and a few systems have exploited this fact (e.g., [10], [2]). GeoPaxos differs from existing systems in the way it implements partial order. GeoPaxos uses multiple independent instances of Multi-Paxos [11] to order operations—hereafter, we call an instance of Multi-Paxos a Paxos group or simply a group. Operations are ordered by one or more groups, depending on the objects they access. Operations ordered by a single group are the most efficient ones since they involve servers in datacenters in the same region. Operations that involve multiple groups require coordination among servers in datacenters that may be far apart and thus perform worse than single-group operations. GeoPaxos's approach to partial order can take advantage of public cloud computing infrastructures such as Amazon EC2 [12]: fault tolerance is provided by nodes in datacenters in different availability zones, within the same region; performance is provided by replicas in different regions. Although intra-region redundancy does not tolerate catastrophic failures in which all datacenters of a region are wiped out, most applications do not require this level of reliability [1].

Third, to maximize the number of single-group operations, GeoPaxos exploits geographic locality. Geographic locality presumes that objects have a preferred site, that is, a site where objects are most likely accessed. Geographic locality is common in many online services. For example, operations on a user's data often originate in the region where the user is. Some distributed systems exploit locality by sharding the data and placing shards near the users of the data (e.g., [1], [4]). GeoPaxos does not shard the service state; instead, it distributes the responsibility for ordering operations to Paxos groups deployed in different regions. Operations are ordered by the groups in the preferred sites of the objects accessed by the operation.

The rest of the paper is structured as follows. Section II details the system model and recalls fundamental notions. Section III overviews the main paper contributions. Section IV details GeoPaxos. Section V describes our prototype. Sec- tion VI presents our performance evaluation. Section VII reviews related work and Section VIII concludes the paper.

[add better explanations of the things] [add images]

3

GeoPaxos with b+tree

Now that we have discussed the needed background, we can move onto the next step: using a b+tree on top of GeoPaxos to store the data objects. A b+tree has many interesting characteristics that make it particularly suitable for some types of applications, but it also brings some new challenges to the table. The data structure has to be indentically replicated in every replica: this means that all the operations done on the tree have to be deterministic and executed in the right order. While this is not particularly complicated on other data structures, such as a HashMap, this becomes more complicated with a tree, where we have many branches and nodes that may split and change the whole structure of the tree.

Furthermore, the usage of GeoPaxos and a tree brings the need for a new type of operation. In GeoPaxos the objects are assigned to one or more groups, depending on the type, number and origin of accesses. Of these groups, one will also be chosen in each replica to be the preferred partition for the operation, usually based on geographic location. There has to be a moment when these groups are decided and calculated for each object in the b+tree. For this, we have a command called repartition. The command takes the workload of an object, which is the number of reads and writes from each group on this object, and a graph that represents the geographic location of the various replicas. It then calculates the optimal placement of the object in the groups, that with the given workload would give us the minimum average latency. This operation can take be a big performance bottleneck, since it has to be executed for every object in the tree, and since we have to consider every combination of groups it scales exponentially on the number of groups. We therefore want to find a fast way to perform this operation so that we still get the right assignment of objects in a short amount of time.

In this chapter I will first explain what a b+ tree is, how it works and what are its advantages and disadvantages. I will then describe the specific b+ tree used in our application. Then I will go over the various approaches that were attempted to improve the performance of the repartition optimization, followed with tests on the performance of the repartition only and finally with GeoPaxos as well.

3.1 B+ tree

[maybe say first what a b-tree is]

A B+ tree is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves.[1] The root may be either a leaf or a node with two or more children.[2]

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node,[1] typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

The order, or branching factor, b of a B+ tree measures the capacity of nodes (i.e., the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as m, is constrained for internal nodes so that $\lceil b/2 \rceil \leq m \leq b$. The root is an exception: it is allowed to have as few as two children.[1] For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least $\lceil b/2 \rceil$ and at most b. In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary and at most b-1.

[add characteristics from wiki]

Insertion[edit] Perform a search to determine what bucket the new record should go into. If the bucket is not full (at most b-1 entries after the insertion), add the record. Otherwise, before inserting the new record split the bucket. original node has $\lceil (L + 1)/2 \rceil$ items new node has $\lfloor (L + 1)/2 \rfloor$ items Move $\lceil (L + 1)/2 \rceil$-th key to the parent, and insert the new node to the parent. Repeat until a parent is found that need not split. If the root splits, treat it as if it has an empty parent and split as outline above. B-trees grow at the root and not at the leaves.[1]

[explain the statistics, how/when they're upadted, node splitting, aging]

3.2 Optimization approaches

3.2.1 group by level(bucket)

[pic of grouped levels]

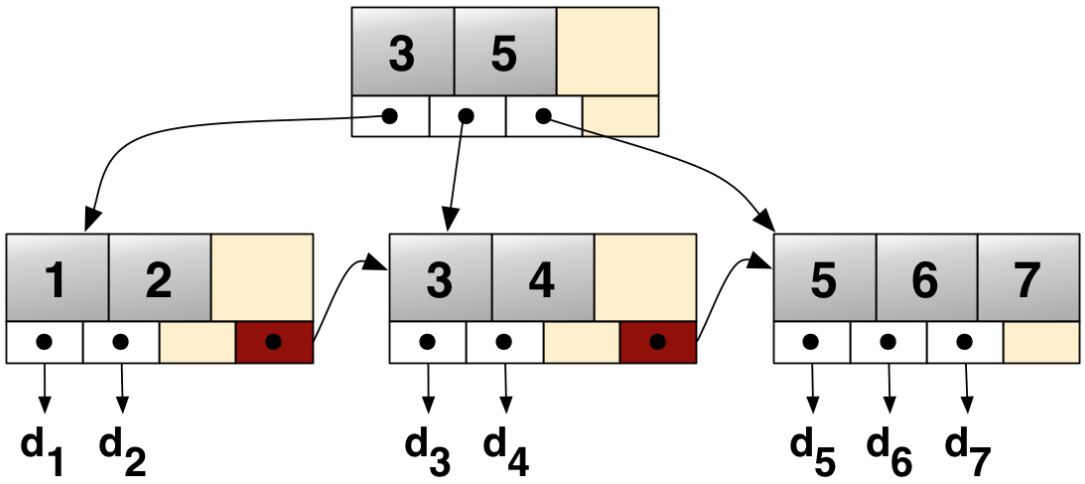

Figure 3.1: The architecture of the system. The *Viper IDE* and *Viper Debugger* boxes denote, respectively, the main Viper extension and the new debugger extension. They both interact, independently, with Visual Studio Code. Viper Server is responsible for running the verification backends.

3.2.2 dynamic bucket

if few reads compared to parent, inherit [pic of grouped levels]

3.2.3 hot groups

3.2.4 LRU caching

3.3 Optimization tests

3.4 GeoPaxos tests

[say how the clients act, put a skew graph, different types of clients, repartitino timings...]

4

Conclusion

4.1 Future work

Bibliography