Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Store*d*: A Distributed Immutable Blob Store

Author: Thomas de Zeeuw (2599441)

1st supervisor: Animesh Trivedi
2nd reader: Alexandru Iosup

A thesis submitted in fulfilment of the requirements for the joint VU-UvA Master of Science degree in Computer Science

December 11, 2020

Abstract

In today's computing a lot of data is stored: storage is getting cheaper, individual files are getting larger and the volume of data stored is increasing. [1] predicts that the data stored globally will grow from 33 Zettabytes in 2018 to 175 Zettabytes by 2025.

To store and use this data more efficiently new databases changed from the commonly used relational (SQL) model to purpose-designed models. These new databases explore different structures to store and expose data, examples include key-value, wide-column and graph databases.

However there is another axis on which data can be differentiated: mutable versus immutable data. This immutability of data provides opportunities that stores designed mutable data can't take advantage of. Store*d* is a distributed immutable blob store designed to take advantage of the opportunities that immutability provides.

Store*d* is evaluated and compared against Redis and etcd. However the results are not conclusive enough to say that taking advantages of the immutable data properties leads to a faster or simpler distributed data store.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

In today's computing a lot of data is stored: storage is getting cheaper, individual files are getting larger (e.g. higher resolution images and videos) and the volume of data stored is also increasing as more users make extensive use of online services. [1] predicts that the data stored globally will grow from 33 Zettabytes in 2018 to 175 Zettabytes by 2025.

To store and use this data more efficiently new databases changed from the commonly used relational (SQL) model to different purpose-designed models. These new databases, often referred to as NoSQL [2] databases, explore different structures to store and expose data. Column stores are an example of such a NoSQL database. Column stores store data in columns rather than rows (which is usually the case for SQL databases) to efficiently retrieve a single property from many entities, rather then retrieving all/most properties of a single entity (common in SQL). A well known implementation of a column store is Cassandra [3]. A radically different model is that of Graph stores, such as Neo4j [4]. The model exposed by graph stores allows for more efficient storing and using of relations between objects, e.g. the relation between two people on a social media website. A last example is a key-value store, such as Redis [5], which exposes a simpler model compared to SQL. It only allows storing values at a key location, not supporting structured data or complex access patterns to the data.[1]

However there is another axis on which data can be differentiated: mutable versus immutable data. Some data is inherently immutable, examples include images and videos (stored on social media websites or on services like Netflix or YouTube) and source code for library managers (such as Cargo [6] and npm [7]). This immutability of data provides various opportunities that stores designed for dealing with mutable data can't take advan-

[1]Note that Redis has grown beyond a simple key-value store and now supports more complex structures and access operations.

tage of. This thesis explores opportunities and the design space of immutable distributed storage.

1.2 Research Questions

The main research question for this thesis is: *does taking advantage of the immutable property of data lead to a better distributed (immutable) data store?* Since this a very broad question it will be split into three questions.

First, RQ1 *does taking advantage of the immutable property of data lead to a simpler design of a distributed (immutable) data store?* I theorise that using the immutable property of data leads to a simpler design of a distributed store. When using mutable data there needs to be some kind of coordination/consensus between operations (such storing data at a certain location, be it a key or table row) to ensure strong data consistency. Two commonly used algorithms for distributed consensus are Paxos [8] and Raft [9], both are considered complex to implement.[1] I theorise that immutable data stores can use a simpler algorithm to implement consensus, while maintaining similar performance and the same consistency guarantees.

A store can also choose to be weak or eventual consistent, meaning that the order in which events are visible to the clients is not consistent. However this thesis (and prototype) implements a strongly consistent store as the author is of the opinion that pushing the consistency difficulties to the application layer is a bad idea, also see papers such as [10, 11]. Furthermore weak consistency is often chosen for performance reasons, something immutable data could help with as per the next research question.

The second research question is: RQ2 *does a distributed store designed for immutable data perform better than stores designed for mutable data?* Here the term *perform better* need additional clarification, for this question I'll be looking at two performance characteristics of a store: latency and throughput. I theorise that a store designed for immutable data can reduce the amount of blocking/locking/coordination in (all kinds of) operations, allowing for more parallelism.

The third research question is: RQ3 *does a distributed store designed for immutable data scale better than stores designed for mutable data?* Although similar to the second question about performance this looks at the store from a scalability perspective. This question looks at if adding nodes to the store reduces the performance of individual operations, e.g. what is the performance (as measured in RQ2) of a store operation for the store with three nodes compared to five or ten nodes. The hypothesis being that the latency would increase (need to coordinate/reach consensus with more nodes), but the throughput would increase (more nodes to handle the requests).

[1]In fact Raft was created because "Paxos is quite difficult to understand" [9].

1.3 The Case For Store*d*

To answer the research questions I needed to compare a data store designed for mutable data to a store designed for immutable data. There are many stores available that are designed for mutable storage, some of which are are mentioned in the related work chapter 6. However there weren't that many immutable data stores. Facebook's f4 [12] is a distributed immutable blob store, however it is proprietary software and thus wasn't available to me. LinkedIn's Ambry [13] is open source, but is desgined LinkedIn's Petabyte (PB) scale. Finally I found Ringo [14], which claims to have a similar design to Amazon's Dynamo [15], but is unmaintained as the last commit was twelve years old at the time of writing. None of these stores were a good match to answer the research questions. In agreement with my supervisor we decided that I would design and build a prototype to answer the research questions, making this thesis an experimental system research thesis.

To lend some credibility to the usefulness of an immutable blob store I refer to LinkedIn's Ambry [13] and Facebook's f4 [12], both are distributed immutable blob stores that work at PB scale. Both Ambry and f4 have different kind of nodes (e.g. front-end and storage nodes), which makes sense for their designed scale, but do make them harder to deploy.[1] However there are also use cases that don't deploy at a PB scale, but could use a immutable data store. For these use cases there is no small-medium scale store that can easily be deployed, using a single binary and a simple configuration, similar to what Redis [5] or PostgreSQL [16] offer in the mutable store space. Examples of those use cases include:

- Library or container registries, such as Cargo [6], npm [7] or Docker Hub [17]. These registries store libraries (source code) or containers images that must be immutable. If you need version x of a library you always want the same version to ensure your code works correctly. These kind of registries see a read heavy (99%+) workload.[2] The size distribution of the uploaded content is not available, however for libraries this is *generally* under 100 kB.[3]

- Hosting static files such as videos or images. Not all websites operate at the scale of Facebook or LinkedIn, but many have user uploaded static images (e.g. to show with a user's profile). For these kinds of applications the workload is also read heavy, e.g. Ambry claims >95% read traffic [13]. The sizes of these files vary greatly.

[1]Compared to deployments consisting of a single binary. For example there is no question of how many front-end vs. storage nodes are needed.

[2]npmjs.com currently supports 1.3 million packages (insert requests) versus 75 billion downloads a month (read requests) [18]. crates.io has over 4.5 billion downloads (read), with just over 300 thousand published crates (including all versions, insert requests) [6]. Both figures indicate a workload of 99%+ read operations.

[3]The is an assumption made by the author based on using both npm and crates.io.

- Machine learning data sets. The data sets used in machine learning are immutable (while the system is learning), but these sets are not always at a PB scale. The read/write ratio for this kind of workload vary depending on the learning implementation.

This thesis targets those GB-TB scale use cases,[1] building a distributed immutable blob store (prototype).

1.4 Contributions Made

The main contributions made in this thesis are the following.

First, this thesis shows that there is potential in considering the mutable/immutable aspect of data when optimising and improving stores, similar to how graph databases or key-value stores are optimised for the structure of the data they store. This idea is not new, but this thesis shows there is potential of additional benefits (such as performance) at small to medium scale, where stores such as f4 [12] and Ambry [13] already did this for larger scale stores. However this thesis does not provide conclusive evidence that designing for immutable data makes for a *better*[2] store, the evaluation results were not good enough for that.

Second, the prototype Store$d$. Store$d$ is a prototype created to evaluate the research questions described in section 1.2. The code is open source and available at [19].

Third, as a part of Store$d$ I developed the Heph framework. Heph is an actor framework for the Rust programming language [20] based on asynchronous functions. It is also open source and available at [21].

1.5 Thesis Outline

This thesis continues with the following chapters:

- Chapter 2 gives an overview and description of the design of Store$d$.
- Chapter 3 describes Store$d$s implementation in detail.
- Chapter 4 evaluates Store$d$, describing the experiments and results used to answer the research questions described in section 1.2.
- Chapter 5 contains some further discussions and an experience report of my thesis. It also discusses future work in both immutability (in data stores) and Store$d$.

[1]The prototype should also work at a smaller and larger scales, but those scales are not the focus of the prototype.

[2]For a definition of "better" see section 1.2

- Chapter 6 discusses related work.
- Chapter 7 finishes the thesis with a conclusion.

Chapter 2

Design

This chapter describes the design of Store*d*. It starts with an overview and general description of the store, followed by the desired system properties, assumptions and limitations of the design. Next are the design decisions regarding the distributed functionally, followed by a description of the supported kinds of requests. Chapter 3 describes how this design is actually implemented in the Store*d* prototype.

2.1 Overview

Store*d* is a distributed, immutable blob store, it supports three requests: inserting, reading and deleting Binary Large OBjects (or *blobs*), a single entity seen an opaque sequence of bytes. Clients can store blobs by providing the bytes that make up the blob. Store*d* will calculate a key for the blob, which is returned to the client as response. Reading or deleting the blob is as simple as providing the key returned by the store request.

In key-value stores the client can specify the key for the value stored, however writing to a key once and never being able to modify it would require the client to determine some kind of scheme to generate unique keys. Store*d* instead generates the key for the client. I decided on using the SHA-512 (as specified in FIPS 180-4 [22]) checksum of the blob. Using SHA-512 has two advantages. First, SHA-512 is a deterministic algorithm ensuring that the same blob won't be stored twice, as that would be a waste of storage space. Second, since the key is a checksum of the blob it can be used to validate the correctness of the blob, catching corruptions on disk and in transit.

Clients interact with Store*d* using HTTP/1.1. I choose HTTP because it's a well known and widely used protocol. Furthermore it helps with deployment of Store*d* as any existing HTTP tool (of which there are many) can be used with Store*d*. For example PostgreSQL [16] has its own custom protocol for clients, however this means that if a proxy is required it needs to be purpose build software, e.g. PgBouncer [23]. For Store*d* any existing HTTP proxy can be used, such as nginx [24] or HAProxy [25].

2.2 System Properties

I decided that Store*d* should be strongly consistent. This means that all nodes are in a consistent state; if a blob is available on *one* node it should be available on *all* nodes. To achieve strong consistency it's required to use synchronous replication. Compared to weak consistent or eventual consistent stores this means that there is no "gap" (in time) in which the blob is available on one node, but not on another. With asynchronous replication, if the node that accepted (and responded to) the insert request crashes in this "gap", the blob would be lost. Another argument for providing a strongly consistent interface is simplicity for the clients, e.g. Google's Spanner was changed to a stronger consistency based on developer experience [10].

Ambry and f4 don't have the same properties as Store*d*. Both stores replicate to multiple (geographically dispersed) data centres adding latency the replication process. For performance reasons both stores use asynchronous replication. Asynchronous replication does prevent both stores from achieving strong consistency. Store*d* doesn't improve the performance of multi datacentre replication, but it's not targeting this use case.

2.3 Assumptions

The design of Store*d* has a single assumption. All nodes can be connected to all other nodes to create a fully connected network. Note that the *implementation* of Store*d* has more assumptions.

2.4 Limitations

Like most designs Store*d* has a number of limitations in its design I would like to mention upfront. For simplicity I decided that all nodes need to store all blobs. This has two implications.

First, it means that the storage requirement grows with the number of nodes it uses. In most cases this is overkill once more than three nodes are used as having a blob stored in three different locations is usually seen as acceptable.[1] Store*d* wastes a lot of disk space storing all blobs, but this does reduce complexity. Because solving this limitation doesn't answer any of the research questions I decided not to solve it. Possible solutions to this limitation are discussed in section 5.4.

[1]Both Ambry and f4 store blobs in three data centres [12, 13].

Second, as all nodes store all blobs, all of them need to participate in and commit to the consensus algorithm. The consensus algorithm is required to keep all nodes in a consistent state, see section 2.2. As all nodes need to partake and commit to consensus, this means a single peer failure will block all insert and delete requests. Like with the excessive storage usage, solving this limitation doesn't help answering the research questions, so I decided not to solve it. Section 5.4 describes possible solutions to this limitation.

2.5 Distributed Design Decisions

For simplicity I decided that each node must be connected to all other nodes, creating a fully connected network as shown in figure 2.1. Requiring a fully connected network simplifies the implementation as each node can directly communicate with all other nodes.

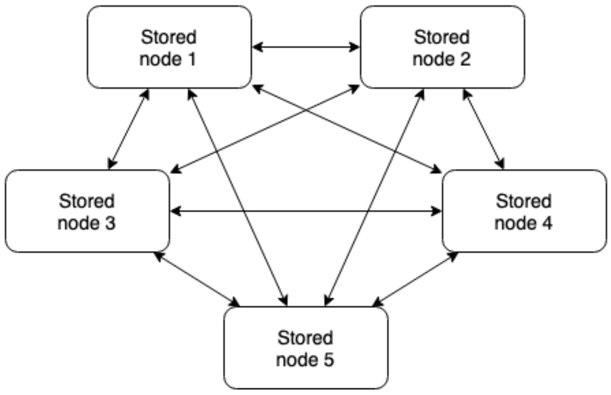


Figure 2.1: Example of Store*d*'s fully connected peer network for five nodes.

To ensure all nodes remain in a consistent state we need to achieve consensus between nodes, for which we need a consensus algorithm. There are many options available, such as Paxos [8], Raft [9] and Two Phase Commit (2PC) [26]. I decided to go with the Two Phase Commit (2PC) protocol because of its simplicity. However the 2PC protocol has a number of known limitations.

One of these limitations is that it uses a single node as coordinator, through which all insert and delete requests must go through. This coordinator thus becomes a single point of failure. However as long as causal ordering [27, 28] can be maintained on all requests this isn't required, allowing multiple coordinators to process insert and delete requests concurrently.[1] Store*d* imposes causal ordering on all requests at node level and uses multiple concurrent coordinators.

Another limitation is that its a blocking algorithm, locking out all others queries (a single

[1]Section 3.3.2 in the implementation chapter describes how Store*d* achieves causal ordering for all requests. Putting this information in this design chapter would involve too many implementation details.

run of the 2PC algorithm) while the coordinator runs a query. However since we can use multiple concurrent coordinators, as per above, this is isn't a problem.

2.6 Request Handling

Store*d* supports three kinds of requests: inserting, reading and deleting blobs, all are described in the following sections. Note that these sections only give a high-level overview, a detailed description of the implementation (including handling failures) is provided in section 3.5.

2.6.1 Inserting Blobs

Inserting blobs has a very simple API: a HTTP `POST` request to the `/blob` URL, with the blob provided as HTTP body. The requests is processed following the steps below.

1. The node accepting the request stores the blob locally, checking if its not already stored.
2. The node that accepted the request, now acting as coordinator for the 2PC query, asks all others nodes to store the blob.
3. Once all nodes agree to store the blob they all commit to storing it, making it available for reading.
4. Finally a response is returned to the client, which includes the `Location` HTTP header with the URL where the blob can now be accessed.

2.6.2 Reading Blobs

Reading a blob can be done by sending a HTTP `GET` request to the `/blob/$key` URL, where the `$key` was returned by the insert request in the `Location` header. The blob is retrieved from local storage and returned to the user. Note that in the implementation causal order is still maintained even though no peer communication is involved, the details of this are described in section 3.3.2.

2.6.3 Deleting Blobs

Deleting blobs can be done by sending a HTTP `DELETE` request to the `/blob/$key` URL, where the `$key` was returned by the insert request in the `Location` header. The request is processed in a similar way as that of an insert request.

1. The node accepting the request prepares its local storage, checking if a blob is stored at the key location.

2. The node that accepted the request, now acting as coordinator for the 2PC query, asks all others nodes to prepare their local storage to delete the blob.
3. Once all nodes agree to delete the blob they all commit to deleting it, making it unavailable for reading.
4. Finally a response is returned to the client.

Chapter 3

Implementation

This chapter describes the implementation of Store*d*, which is based on the design found in chapter 2. It starts with the assumptions made in the implementation, followed by the limitations. After that a high-level overview is given of the implementation, as well as a detailed description of the storage layer. Finally all processes are described in detail in the last sections of this chapter.

3.1 Assumptions

In the design of Store*d* I've made a single assumption: all nodes can be connected to all other nodes. However the implementation of Store*d* makes some more assumptions.

First, it assumes that all nodes respond to network requests within some bounded time. If the node does not respond within this bounded time they are considered as failing/failed and will vote to fail (abort) the 2PC query.

Second, it assumes that the file system is correctly implemented according to its manual. For example Store*d* relies on `fdatasync(2)` to flush data and the file's length to disk and survive power failures. If the file system fails to do this, without returning an error, Store*d* will not be able to recover from power failures and deliver on its fault tolerance promises.[1]

3.2 Limitations

Currently Store*d* only runs on Unix OSes with a focus on Linux, but it could be ported to non-Unix OSes such as Windows.

A bigger limitation is the dependency on the OS and its page policy/usage. The implementation makes heavy use of `mmap(2)` (see section 3.4.3), which depends on the OS

[1]Note that in the past `fsync(2)` has had it share of issues [29], I believe this problem should be solved in the OS/file system, not in all applications using them, and are thus out-of-scope for Store*d*.

for caching of and optimal write-back to the data file. This introduces undeterminism in both reading and inserting blobs. To alleviate this Store*d* uses `madvise(2)` to inform the OS about the intended usage of the mapped memory pages, but this doesn't remove the limitation.

3.3 Overview

Store*d* can run in one of two modes: stand-alone mode or distributed mode. If Store*d* is only running on a single node, i.e. not connected to any other nodes, it's running stand-alone mode. In stand-alone mode the consensus algorithm will not run as only one node determines what is and isn't in the store, thus no consensus is needed.

When Store*d* is running in distributed mode it will be running on two or more nodes, all running the same binary. Each node is connected to all other nodes, creating a fully connected network. Once two nodes are connected they consider each other as *peers*.

3.3.1 Single Node

Figure 3.1 shows an overview of the architecture of Store*d* at node level. Store*d* is build on *Heph* [21], an actor framework I've build in the Rust programming language [20]. Heph (and thus Store*d*) uses the *actor model* [30], in which an *actor* is the main component. An actor can receive messages from its own inbox, in response to which it can do computation on its local memory, create more actors and send messages. Because actors can only modify their own private state it doesn't need to synchronise (by using locks or atomic instructions etc.) on this memory [31].

Each node has a single *database actor* that is responsible for serialising access to the storage layer. The database actor is implemented as a synchronous actor in the Heph actor framework, meaning it runs on its own thread and uses blocking operations for disk I/O. The database actor has unique access to the database files so it uses no locking or any other kind of synchronisation itself.

The decision to make a single actor (thread) control the access to the database files and related in-memory structures is based on two factors. First, it simplifies the implementation. Actors that accept incoming client requests (see below) use asynchronous network I/O, mixing that with blocking I/O is pitfall for poor performance.[1] For example a seemingly simple system call such as `open(2)` can still block the thread [34]. This means I

[1]Note that although asynchronous disk I/O exists it has a number of problems. For example `aio`, the POSIX interface for asynchronous I/O, is implemented in user-space (glibc) on Linux [32] and only works with `O_DIRECT`, bypassing OS cache. More problems are described in the introductory paper of `io_uring` [33], a new Linux interface for asynchronous I/O. `io_uring` looks promising, but I decided against using it because it's Linux only and requires a new kernel version (5.1+), which was not commonly deployed at time of starting work on Store*d*.

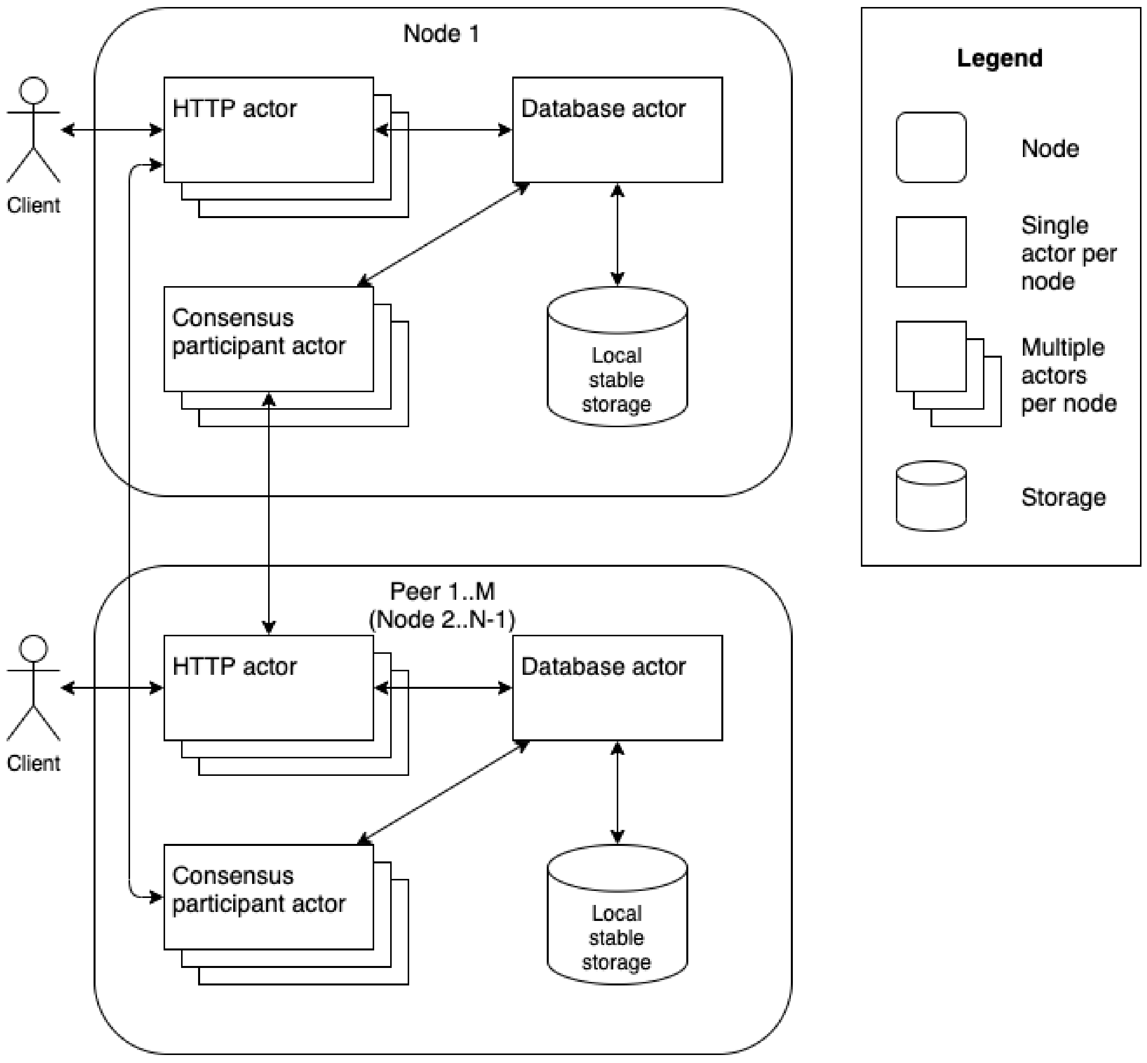


Figure 3.1: Store*d* architecture (showing two nodes).

would have to add some kind of off-loading for the blocking I/O. Replacing this with a (non-blocking) Remote Procedure Call (RPC) simplifies the implementation. Second, there are a number of stores that only use a single thread for disk access, showing that using single thread for disk access is not a limiting factor: [35, 36, 37].

I did not have enough time to properly evaluate the performance implication of the decision make a single actor control the access to the datastore files.

Incoming HTTP requests are handled by a *HTTP actor*, of which one is started for each incoming TCP connection. On a single TCP connection multiple HTTP requests can be send as HTTP pipelining is supported. The HTTP actor is an asynchronous actor, meaning that it runs on a thread multiplexed with other asynchronous actors, sometimes referred to as a *coroutine* or *green thread*. To not block other actors it only uses asynchronous I/O for handling the HTTP requests. All HTTP actors have a communication channel to the database actor, allowing them to communicate (sending messages, RPC etc.) with the database actor.

Finally each node runs zero or more *consensus actor*s that act as a participant in the 2PC algorithm. A new consensus actor is started each time the 2PC algorithm is started by a coordinator and requests participation of the node. The consensus actor behaves similar to the HTTP actor, but handles requests on behalf of its peers rather than its clients (the end-users of Store*d*).

3.3.2 Achieving Causal Ordering

As described in section 2.5 of the design chapter, to support multiple concurrent 2PC coordinators it is required to ensure causal ordering [27, 28] for all requests. In Store*d* this is achieved by using the order in which the database actor receives messages from its inbox.

To demonstrate (but not proof) the correctness of causal ordering consider the following example. Let us have three nodes: A, B & C. In parallel nodes A and B receive an insert request for the same blob, the desired result of which is that we only insert the blob once with all nodes agreeing on it's metadata (such as the created at timestamp). Both node A and B follow the process as described in section 3.5.2, assuming the processes ran in parallel at some point both nodes want to commit to their version of the blob. This is where the order of the database actor's inbox comes in to ensure causal ordering. The database actor receives the messages to commit to inserting the blob in *some* order, as it can only receive a single message at a time. This ensures that the first received query will succeeded and the second message will fail (as the blob is already inserted at that point), providing us with causal ordering of the two insert requests.

3.4 Storage Layer

The database on disk is controlled by a single database actor as described previously. It uses no locking or other synchronisation methods as that is not required as it's only accessed from a single thread.

3.4.1 On-Disk Representation

The database is split into two files on disk (stable storage): the data file and the index file. The idea of splitting the storing of blobs (values) from the keys in different files is taken from the paper WiscKey: Separating Keys from Values in SSD-conscious Storage by Lanyue Lu et al [38]. Figure 3.2 shows the layout of the two files on disk.

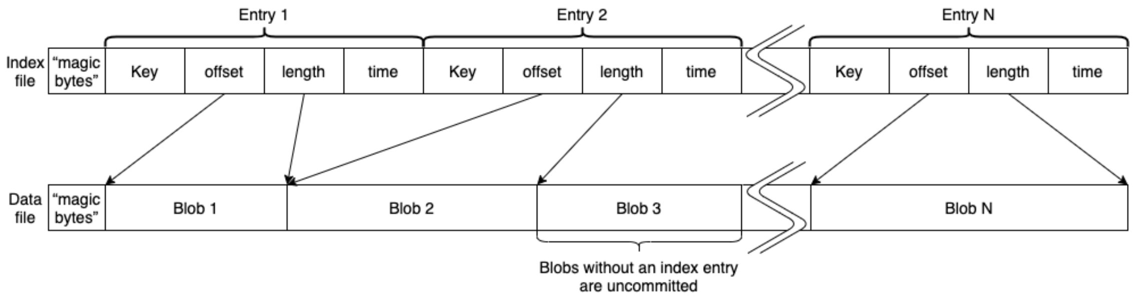


Figure 3.2: On-disk representation.

The index file keeps metadata about the blobs stored *in* the datastore. The index file starts with "magic" bytes, which serves as a sanity check to ensure we've opened a correct index file. Next it contains zero or more *entries* (88 bytes each), each one describing the metadata of a single blob. Each entry contains the fields described in table 3.1.

key	The SHA-512 checksum of the blob (64 bytes), used to read or delete the blob.
offset	Offset into the data file (8 bytes), which is the starting point of the blob.
length	The length of the blob (4 bytes), together with the offset this gives the range of bytes (in the data file) that make up the blob.
time	Either the 'created at' time or the 'deleted at' time (12 bytes). The time is inserted as an *unsigned 64 bit integer* representing the number of seconds since Unix epoch (Jan 1970) and an *unsigned 32 bit integer* representing the sub-second nanoseconds. There are 1 billion nanoseconds in a second, which can be represented using only 30 bits, which leaves two bits unused. One of those bits is used to indicate whether it's a 'created at' or 'deleted at' time. The other bit is used to indicate that the blob is no longer stored in the data file, this is used when the blob is deleted and the blob's bytes are removed from the data file.

Table 3.1: Index entry fields.

In the data file the bytes that make up all the blobs are stored. Just like the index file it starts with "magic" bytes. After that the data file contains no metadata or structure of any kind as all of that is stored in the index file, it's just all blobs appended in a single file.

The index file determines what blobs are actually *in* the datastore, this means that even though a blob might be in the data file it doesn't mean that those blobs are accessible. As a result the data file might contains bytes that are not used, for example when a blob's bytes are added to the data file, but is never committed (e.g. in case of a crash) and thus no index entry is ever created. The index file always has the final say in what blobs are and aren't *in* the datastore.

Neither the data or index file is encrypted. This is a deliberate choice as *if* encryption is wanted it should be applied to the entire disk, and thus falls out of scope for Store*d*. In the author's opinion encryption is a task of the file system, not individual programs.

3.4.2 Storage Durability

Inserting a blob to the datastore is a two step process, following a similar structure found in the 2PC algorithm. This two step process ensures the blob is durably stored on disk.

The first step is adding the blob to the data file. This is done by allocating additional space in the data file (if needed) using `fallocate(2)` (or `ftruncate(2)` on OSes that don't support `fallocate(2)`). For performance reasons we allocate at least 65kB of data so we don't have to allocate additional space for each (small) blob inserted. After the additional space is allocated it's mapped into memory using `mmap(2)` and the blob is copied to this file-backed `mmap` area. Finally we start the syncing process by calling `msync(MS_ASYNC)`,[1] but don't wait for it to complete. If Store*d* would crash at this point the blob would not be in the datastore as there would be no index entry pointing to this blob.

Store*d* has an optimisation for inserting blobs larger than a single page (commonly 4kB). Instead of reading the entire blob into memory from a client socket and then copying it to the `mmap`-ed area, it creates a pre-allocated `mmap`-area for the blob to be written into. The actor handling the client's request, e.g. the HTTP actor, can than read from the socket directly into this pre-allocated `mmap`-ed area backed by the data file. This avoids a copy of the blob in user-space.

The second step of inserting a blob is creating the index entry for the blob. This starts with syncing the blob, already written to the file-backed `mmap` area in step one, to disk using `msync(MS_SYNC)`, waiting for it to complete. Next an index entry is written using `write(2)` and synced using `fdatasync(2)`. At this point the blob is durability stored, accessible even after a crash.

3.4.3 In-Memory Representation

The in-memory representation of Store*d* is shown in figure 3.3. At the top is the `Storage` structure, which holds two `HashMap`s (found in Rust's standard library), one for the stored blobs and one for the uncommitted blobs, and the `Data` and `Index` structures. The database actor has unique access to this structure, this way no locking or any kind of synchronisation is required to access it.

The `Data` structure holds one or more `mmap`-ed areas backed by the data file and the file descriptor of the data file. At startup the entire data file is mapped into memory using `mmap(2)`. Once more blobs are added the data file is grown using `fallocate(2)` (as

[1]On Linux, at the time of writing this, this is a no-op.

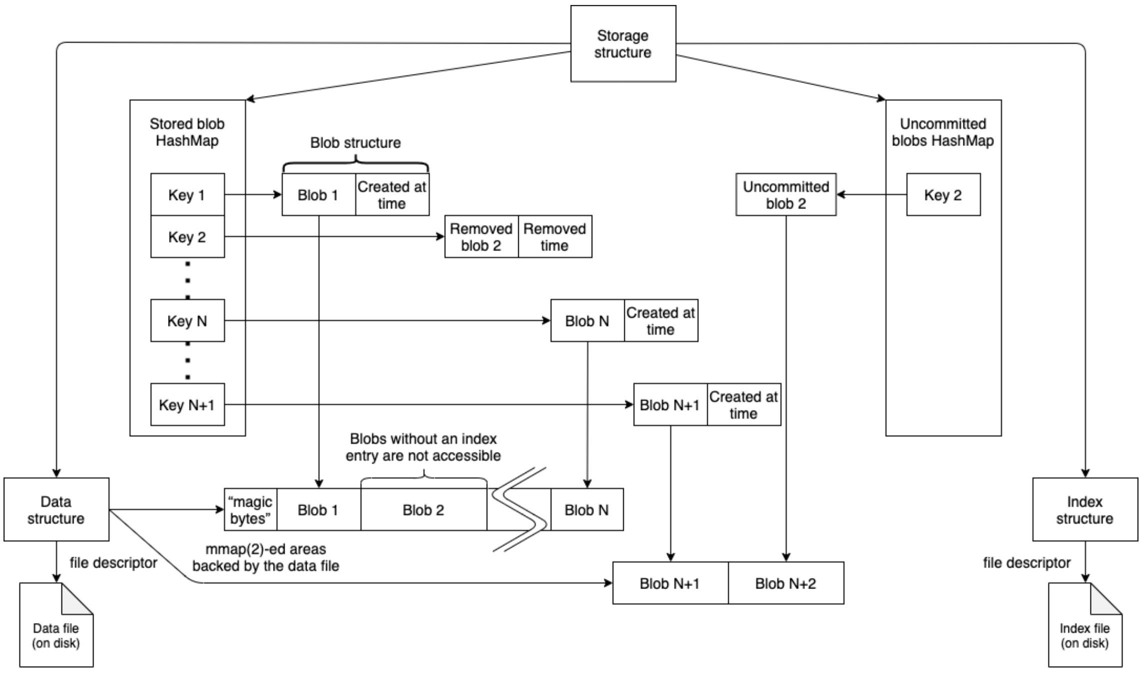


Figure 3.3: In-memory representation.

described previously) and the additional space is also mapped in memory. If however the current area can't be grown, i.e. if the page after our `mmap`-ed area is already used, we need to create another area for the unmapped bytes (leaving the original area in place).

Mapping all blobs in memory has the advantage of not needing to keep all blobs in memory, as that would limit the amount of blobs we can store. The OS will only load the pages from disk (reading the data file) when they are accessed, evicting them when they are not accessed. This allows Store*d* to use the OS cache for both reading and writing without having to deal with all the complexities of that itself.

But this also has a disadvantage: non-determinism of accessing the memory mapped blobs as the blob might not be in memory. If a blob is accessed that is not loaded into memory, it causes a page fault and forces the OS to read the blob from disk, blocking the thread from further execution, before continuing.

To minimise page faults Store*d* uses `madvise(2)` to inform the OS about the memory usage of the blobs. However this doesn't fully remove the disadvantage as it's just an advice and the OS is free to ignore it, or with high contention the OS might not be able read data from disk fast enough to avoid page faults.

The `HashMap` of stored blobs maps keys (the SHA-512 checksum of the blob) to a `Blob` structure. The `Blob` structure acts as “smart pointer”[1] to create a slice[2] to the bytes that make up the blob in the `Blob`’s structure memory mappings, using the information stored in the index file. This allows users of the `Blob` structure, e.g. the HTTP actor, to pretend the blob is completely in memory and use simple system calls such `write(2)` to write the blob to a socket. Furthermore it doesn’t require a lot of memory as the OS can evict memory pages Store*d* doesn’t use, i.e. pages where blobs are stored that aren’t accessed.

The second `Hashmap` maps keys to the uncommitted blobs. At startup this will be empty. An entry is added to this map once a blob is added to the data file, but not yet committed (and thus not yet accessible). An uncommitted blob contains the key, the offset in the data file and blob’s length, everything needed to create an index entry when committing.

Finally there is the `Index` structure which holds the index file descriptor. At startup it is used to create the stored blobs `Hashmap`. When committing insert or delete requests an index entry is written to this file, as described in section 3.4.2.

3.5 Processes

This section will describe all processes of Store*d*, including how Store*d* handles faults in those processes. For processes that handle client requests I’ve created diagrams to show how the request is handled by Store*d*. An example of such a diagram is figure 3.4 (inserting blobs, found in section 3.5.2). On the left of the diagram there will be a HTTP client which represents a Store*d* client, connecting over HTTP. On the same height there are multiple actors which are part of Store*d*. The arrows between the different actors represent communication between actors. The rounded boxes surrounding these actors indicate on what node an actor is running, highlighting communication between actors on the same node and communication between nodes.

In a process there are various paths shown by the different colours in the diagrams. The black arrows show the successful path. The blue arrows show an alternative, but still successful, path, for example when inserting a blob: what happens if the blob is already stored. Finally the red arrows show what happens in case of faults, a red cross indicates a crashed actor. If an alternative or erroneous path is taken, the successful path is no longer followed.

Each process consists of multiple steps which are numbered, starting at 1. Some step numbers have a alphabetical extension, this indicates that the two steps happen concurrently, for example steps ‘1a’ and ‘1b’ are run concurrently.

[1]Rust terminology to indicate a type can behave as a pointer, but isn’t as simple as an integer representing a memory address.

[2]A dynamically sized array.

3.5.1 Startup

Store*d* starts with opening the database files and starting a single *database actor* (per node) which is in control of them. The stored blobs `HashMap` is filled based on the index entries in the index file, as described in section 3.4.3.

Next Store*d* needs to create the fully connected network of its peers, as shown in figure 2.1, and synchronise the blobs that are stored. To create the fully connected network Store*d* will connect to all the peer specified in the configuration. However to support expanding and shrinking networks while running, i.e. dynamic horizontal scaling, we'll ask each peer to share its known peers and connect to those also and ask those for known peers etc. This makes adding more nodes to the network easier, as its not necessary to know all currently running peers while starting. Providing a single (running) peer address in the configuration is enough to create the fully connected network.

After a node is fully connected (connected to all peers) it starts the synchronisation process, which ensures that the set of all blobs that are stored (and possibly deleted again) is the same on this node as it is on all its peers. During this process the node will be fully functional when it comes to peer interaction, i.e. it will partake in 2PC algorithm queries such as used in inserting and deleting processes, to ensure we don't miss inserting or deleting any blobs. However it will not respond to client requests, as the node is not in a consistent state yet.

In the first step of the synchronisation process all known keys are requested from all peers, putting them all in a single set. A key is 64 bytes, so with 10 million (10,000,000) blobs stored (including removed blobs) requesting all known keys requires 640 megabytes (10 million * 64 bytes) of data per node.

Next this set of known stored blobs (by the peers) is compared to the blobs stored locally (on disk). Any blobs stored locally but not in the set of known blobs are shared with the peers, requesting that they store the blob. This is required for the recovery process after a full system failure where the known peers are started later (perhaps automatically) from a clean slate. Note that because we keep track of deleted blobs we don't overwrite deleted blobs, undoing the deletion.

All blobs in the known set that are missing locally are requested from the peers. For this a simple partitioning scheme is used (N missing blobs / M peers), but this is far from optional. A better (future) solution would be to provide the length of the blob, along with the key, and use this to make better decision about splitting the workload among the peers. It could be the case that we're asking for a blob from a peer that is not fully synchronised, if this is the case we simply ask another peer for the blob.

Finally once the node is fully synchronised the HTTP server is started to process HTTP requests coming from clients. Note that the creating of the fully connected network and the synchronisation only happen in distributed mode, in stand-alone mode the HTTP server is started immediately.

3.5.2 Inserting Blobs

Section 2.6.1 has high-level overview of the inserting process, this section will describe the process in more detail, including the alternative and erroneous paths. Figure 3.4 shows the process of how Store*d* inserts a blob, the introductory paragraphs in section 3.5 describes how to interpret the figure. To keep the figure readable failing of coordinating node (the node that accepts the insert request) isn't included, instead this is described separately in the next section 3.5.3.

The process begins with a client submitting a HTTP POST request which contains the blob to insert. This request is accepted by a *HTTP actor*, of which one is started for each TCP connection created between a client and Store*d*.

In the second step the HTTP actor sends a *add blob* request to the *database actor*. As described in section 3.4.2, inserting blobs is done in two phases at the storage layer. First the database actor appends the blob to the data file. Second, it ensures the data is fully synced to disk and adds a new entry to the index file, ensuring the index entry is also fully synced to disk. Only after those two phases are completed a blob is stored *in* the datastore and accessible. However we don't execute the second phase in this step, phase two is done in step 12 of the inserting process. After the blob is added to the data file the database actor returns an insert query that can be used to complete phase two of inserting the blob, this is done is step 3. Note that the blob is not yet *in* the datastore, it's not accessible yet.

Shown with the blue steps 3 and 4 is what happens if the blob is already stored in the database. Instead of returning an insert query in step 3, the database actor returns a message that the blob is already stored. Next in step 4 (blue) the HTTP actor returns an OK HTTP response. Note that 'created at' time of the blob remains unchanged as nothing was written to disk.

Shown with the red steps 3 and 4 is what happens if the database actor fails to append the blob to the data file, or for some other reason fails. The HTTP actor will detect that the database actor failed and responds with a HTTP server error to the client and closes the connection.

In step 4, following the normal process again, the HTTP actor starts the Two-phase commit (2PC) protocol [26], in which itself acts as coordinator and all peers are participants. Each participant starts a *consensus actor* which handles the coordinator's messages. First it requests the blob from the coordinator and adds the blob to the data file, as the coordinator node already did in steps 2 and 3. The consensus actor relays this request to the database in step 5. Once the blob is written to disk the database actor returns a insert query in step 6. Once the consensus actor receives the insert query from the database actor it will vote to commit to inserting the blob in the 2PC protocol, this is step 7b. In step 7a,

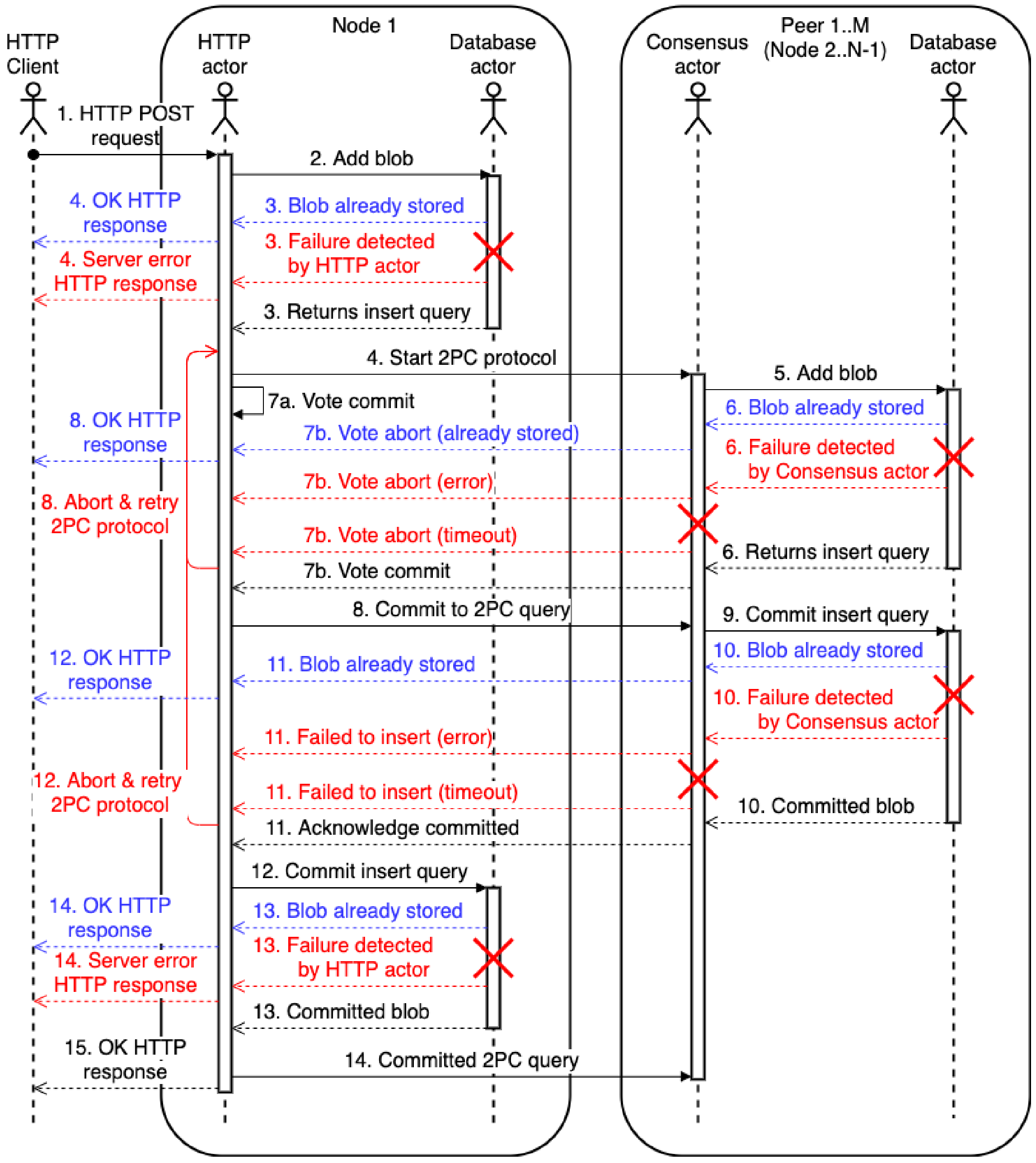


Figure 3.4: Inserting a blob.

which happens concurrently with step 7b, the HTTP actor implicitly also votes to commit to inserting the blob.

Shown with the blue steps 6-8 is what happens if the blob is inserted between the first time we checked in step 2 and when we checked checked in step 5. If one of the participants detects the blob is already inserted, in step 6 (blue), it means the blob is inserted on all nodes (because all nodes store all blobs and are strongly consistent). The HTTP actor aborts the 2PC query and will return an OK HTTP response, similar to what happened in step 4 (blue). The bytes of the blob will be inaccessible and will later be reused if the database is compacted, this is described in section 3.5.7.

Shown with the red steps 6-7b is what happens if one of the participants fails. Similar to step 3 (red), if the database actor on one of the peers fails it will be detected by the consensus actor in step 6 (red) using timeouts. The consensus actor will than respond to the HTTP actor with a vote to abort the 2PC query. Of course the consensus actor can also crash, in this case the consensus actor will not respond at all, for this we also have a timeout which will count as a vote to abort, as shown in step 7b (red).

Depending on the votes for the 2PC query, it will be aborted or committed. If any participant voted to abort the 2PC query it will be retried by the HTTP actor, going back to step 4 as shown by step 8 (red). The 2PC protocol is retried three times before giving up completely and returning a server error to the client.

At this point the limitations of the design show itself: a single peer can block all inserting (and deleting) processes, furthermore the disk usage increases linearly with the number of nodes. These limitations were discussed section 2.4.

If all participants vote to commit to the 2PC query the HTTP actor will send a message to all participants to commit to the 2PC query in step 8. In this message the timestamp at which the blob was officially created is provided to ensure this is the same for all participants. Note that the validity of this timestamp has no influence on the correctness of the process. In step 9 all consensus actors will concurrently commit to the insert query by sending it to the database actor (on their own node). The database actor (on each node) will add a new entry to the index file, ensuring it and the blob in the data file is fully synced to disk, as described in section 3.4.2. Only at this point is the blob stored *in* the database and is it accessible. After the index entry is created the database actor will reply to the consensus actor that the blob is committed. The consensus actor will relay this information to the HTTP actor in step 11.

Shown with the blue step 10 is that the blob could already be stored between step 5 (when the peers checked if the blob was stored before adding it to the data file) and step 9. If one of the participants detects the blob is already stored it will not commit to inserting the blob. Instead the consensus actor will inform the HTTP actor a blob is already stored in step 11 (blue). The HTTP actor will collect the acknowledgements like normal and will respond with an OK HTTP response in step 12 (blue), but it will use the already stored blob.

The red steps 10 and 11 show what happens if the database or consensus actor fails, in either case it will be voted to abort the 2PC query. The HTTP actor will then retry the 2PC query in step 12 (red).

Once the HTTP actor, acting as coordinator, received an acknowledgement of commitment from all participants, itself will commit to inserting the blob in steps 12 and 13. It will then send a committed message to all consensus actors (see section 3.5.3 why this is required) and return an OK HTTP response in step 15.

As before, it could be that the blob is already inserted when committing it in step 12. Shown with the blue steps 13 and 14 is that the HTTP actor will respond with an OK HTTP response, but it will use the already stored blob.

Finally the committing to storage could also go wrong, as indicated by red steps 13 and 14, in which case the HTTP actor will return a HTTP server error.

3.5.3 Handling Coordinator Failure While Inserting Blobs

Figure 3.5 shows the same process for inserting blobs as figure 3.4, hiding certain steps and instead showing what happens if the node acting as coordinator in the 2PC algorithm fails. The numbering for the steps showing the successful path are kept the same as in figure 3.4, as a result the red steps create its own order in some cases.

There are a number of cases that can occur if the coordinator node fails. If the coordinator fails before step 4 (starting the 2PC protocol) we don't have to do anything special as the 2PC algorithm hasn't started. The connection to the client will be (forcefully) closed and the client will see this as an error, this is shown in step 4 (red). The result is that the blob is not inserted.

If the coordinator fails after step 4, but before step 8 (committing of the 2PC query) in the process, the 2PC query will be aborted. The participants will detect that the coordinator has crashed (using timeouts) and will abort the 2PC query as shown in red steps 6 and 7. Like in the previous case: the client will be disconnected and will see this as an error as shown in step 5 (red). The blob is not inserted.

What happens if the coordinator crashes after committing to the 2PC query in step 8 is more complicated. It could be that some of the participants have received a commit message, but others have not. In this case we need finish the second phase of the 2PC protocol using the participants alone. Once a participant detects that the coordinator has crashed it will share this information with all other participants, and whether or not the coordinator send a message to commit to the 2PC query (red steps 11 and 12). If no participants received a message to commit the 2PC query, it will be aborted, as shown by red step 13. From the client perspective it will be if the request failed in an earlier stage

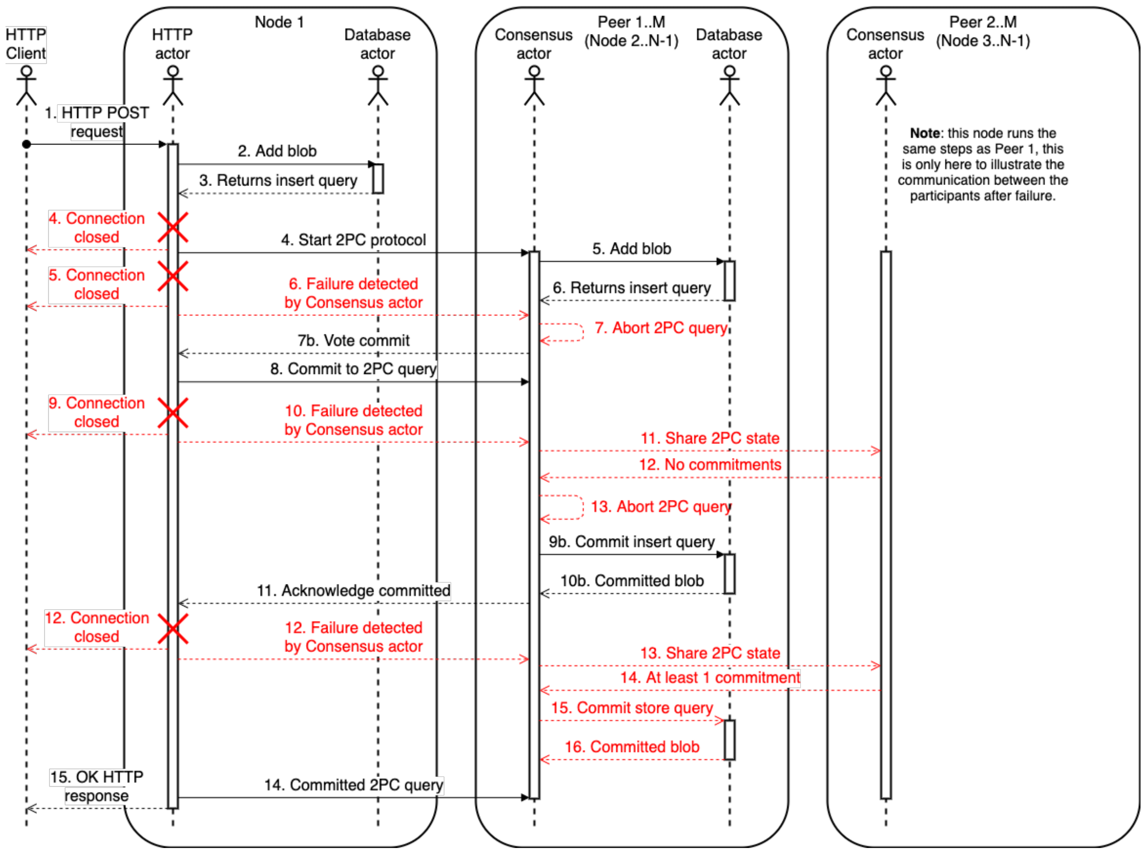


Figure 3.5: Handling coordinator failure when inserting a blob.

of the process, similar to the second case (failing between steps 4 and 8).

If one or more participants committed to the query they share this with all the other participants (red steps 13 and 14), which will also commit to the query as shown by red steps 15 and 16. The result will be that the blob is inserted, but the client will not be informed (as the connection to it has already been disconnected).

This shows why step 14 (of the successful path), the coordinator letting the participant know it's committed to the 2PC query, is needed. This is an indication that all nodes, the coordinator and all participants, are committed to the query and that the above described "participant consensus" is not needed.

3.5.4 Reading Blobs

Figure 3.6 shows the process of how Stored reads a blob from disk. The reading process is very simple and doesn't require communication with any peers. It starts with a client sending a HTTP GET request, which is accepted by a HTTP actor. The HTTP actor than sends a request to the database actor to get the blob in step 2. If the blob is stored in the database its returned by the database actor in step 3, which the HTTP actor relays to the client in the form of an OK HTTP response in step 4. Note when the database actor returns a blob it doesn't read the entire blob into memory, as described in the storage section 3.4. It maps the blob into memory using `mmap(2)` and uses `madvise(2)` to inform the OS we intend to read the memory pages.

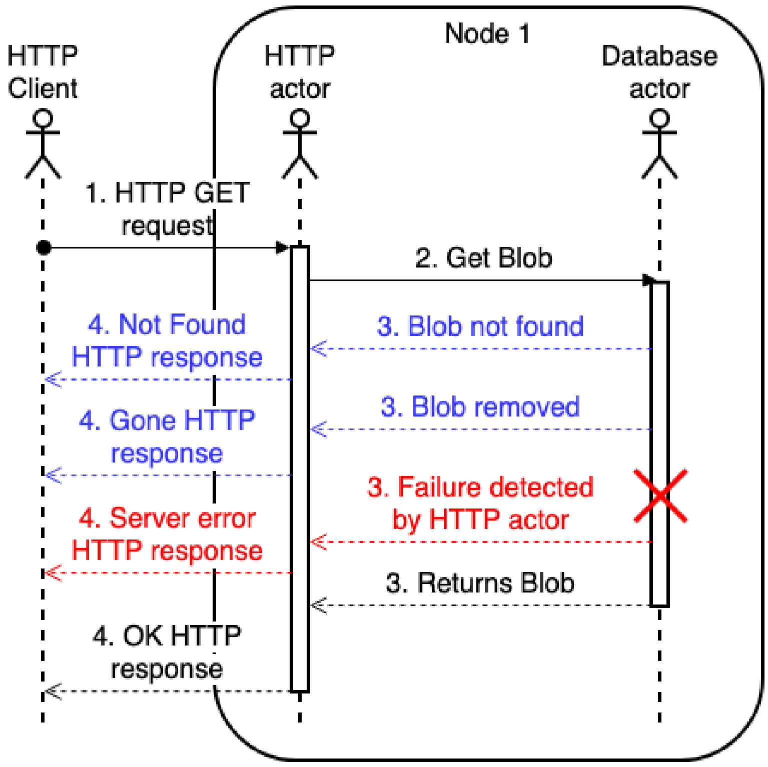


Figure 3.6: Reading a blob.

Shown in blue steps 3 and 4 (the first pair) is what happens if the blob is not stored in the database. The database actors returns this message to the HTTP actor which than responds to the client with a Not Found HTTP response, indicating the blob is not stored.

A secondary path shown by blue steps 3 and 4 (the second pair) shows what happens if the blob was previously stored, but later deleted. In this case the HTTP actor returns a Gone HTTP respond to the client. This allows the client to differentiate from 'blob was never stored' and 'blob was deleted'.

Finally the red steps 3 and 4 show what happens if the database actor crashes or otherwise fails to respond to the HTTP actor. The HTTP actor detects that the database actor failed and will respond with a server error HTTP response to the client.

Similarly to the inserting process, if at any point in the process the node crashes the client will be disconnected and will see this as an error.

3.5.5 Deleting Blobs

The process of deleting a blob is similarly structured as that of inserting a blob, as shown in figure 3.4 and described in section 3.5.2. Figure 3.7 shows the process of deleting a blob. Handling of coordinator failure is described in the next section 3.5.6.

Like in the inserting process, the deleting process begins with a client submitting a HTTP request. The HTTP request method must be DELETE and the key is contained in the URL (part of the HTTP request). This request is accepted and processed by a *HTTP actor*.

Deleting blobs, like inserting blobs, is also a two phase process at the storage layer. First the deleting process must be prepared, ensuring the blob is stored. In the second phase the index entry of the blob is marked as deleted and the changes to the index file are synced to disk. The bytes that make up the blob are unchanged, meaning the data file is untouched in the deleting process. The blob's bytes are removed asynchronously in the compacting process, described in section 3.5.7.

In the second step the HTTP actor sends a *prepare storage* request to the *database actor*, the first phase of the process described above. The preparation will ensure the blob is stored and returns a *delete query* in step 3. Note that at this point the blob is still accessible.

Shown with the two blue steps 3 is what happens if the blob was never stored, or is already deleted. The database actor returns a message that the blob is not stored (anymore). In the two blue steps 4 the HTTP actor receives this message and returns a Not Found (blob was never stored) or Gone (blob was already deleted) response to the client.

Shown with the red steps 3 and 4 is what happens if the database actor fails. The HTTP actor will detect that the database actor failed and returns a server error to the client and closes the connection.

Next in step 4 the HTTP actor starts the Two-phase commit (2PC) protocol, in which itself acts as coordinator and all peers are participants. Each participant starts a *consensus actor* which prepares the storage to delete the blob in steps 5 and 6, same as the coordinator already did in steps 2 and 3.

Once the consensus actor receives the delete query from the database actor it will vote to commit to deleting the blob in the 2PC protocol, this is step 7b. In step 7a the HTTP actor implicitly votes to commit.

Shown with the blue steps 6-8 is what happens if the blob is deleted between the first time we checked in step 2 and in step 5. If one of the peers detects the blob is already deleted, in step 6 (blue), it means the blob is deleted on all nodes (as Store$d$ is strongly consistent). The HTTP actor aborts the 2PC query and will return a Gone response, similar to what happened in step 4 (blue).

Shown with the red steps 6-7b is what happens if one of the peers fails. Similar to step

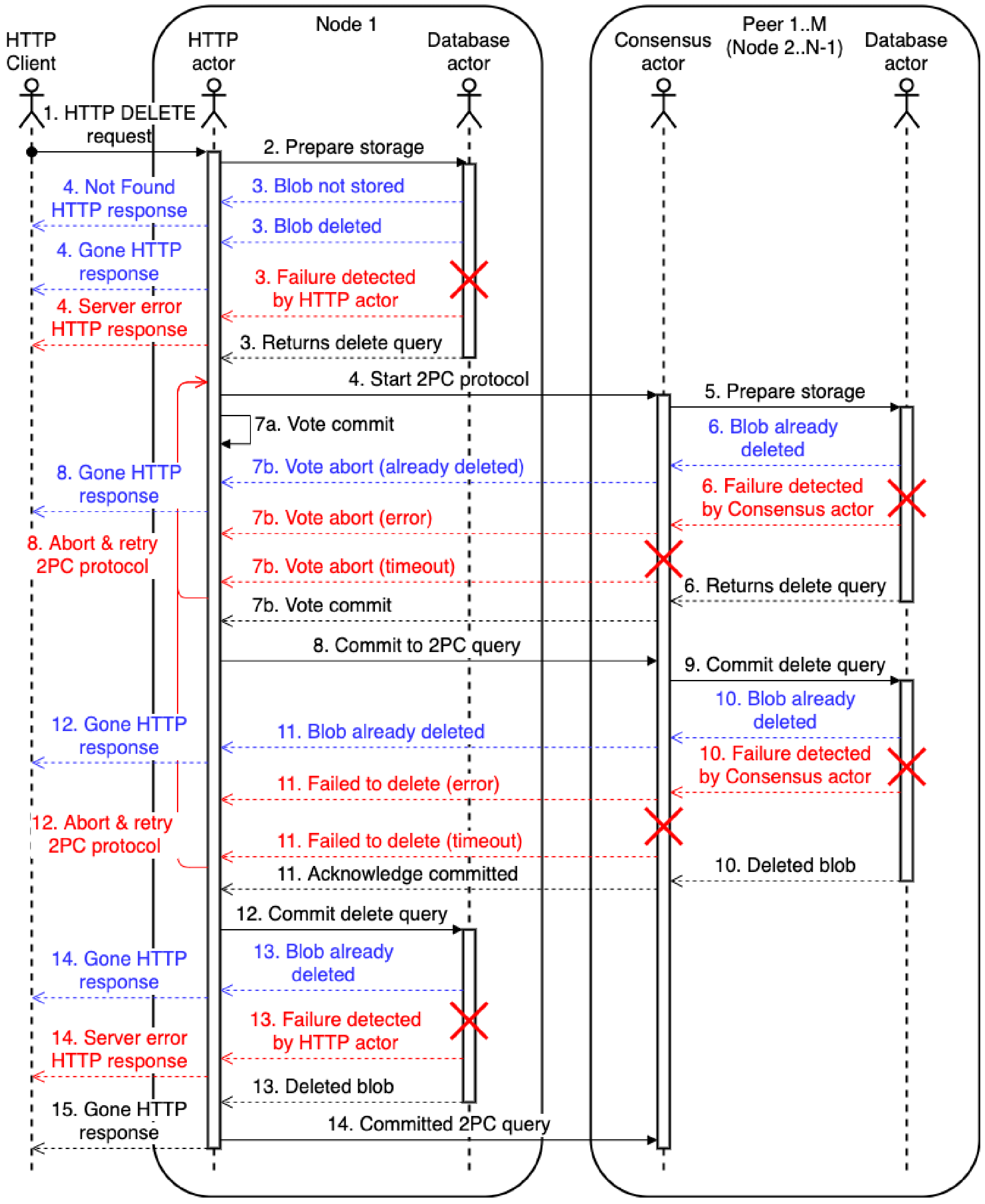


Figure 3.7: Deleting a blob.

3 (red), if the database actor on one of the peer fails it will be detected by the consensus actor in step 6 (red). The consensus actor will than respond to the HTTP actor with a vote to abort the 2PC query. Of course the consensus actor can also crash, in this case the consensus actor will not respond at all, for this we have a timeout which will count as a vote to abort as shown in step 7b (red). Depending on the votes the 2PC query will be retried or committed.

If any participant voted to abort the 2PC query it will be retried by the HTTP actor, going back to step 4 as shown by step 8 (red). The 2PC protocol is retried three times before giving up completely and returning a server error to the client.

If all participants vote to commit to the 2PC query the HTTP actor will send a message to all participants to commit to the 2PC query in step 8. In this message the timestamp at which the blob was deleted is provided to ensure this is the same for all peers. In steps 9, 10 and 11 the consensus actor will commit to the 2PC query and return an acknowledgement to the HTTP actor.

Shown with the blue steps 10 and 11 the blob could already be deleted between step 5 and steps 9. If one of the participants detects the blob is already deleted it will not overwrite the index entry with a new date, instead the old date will be used. The participant will still acknowledge the commitment to coordinator as the end result is the same (the blob is deleted). The HTTP actor will collect the acknowledgements like normal and will respond with a Gone HTTP response in step 11/12 (blue).

The red steps 10 and 11 show what happens if the database or consensus actor fails, in either case it will be voted to abort the 2PC query. The HTTP actor will then retry the 2PC query in step 12 (red).

Once all participants are committed the coordinator will commit to the delete query in steps 12 and 13.

As before, it could be that the blob is already deleted when committing it in step 12. Shown with the blue steps 13 and 14 is that the HTTP actor will respond with an Gone HTTP response, but it will uses the timestamp from the previous deletion (not overwriting it).

Finally the committing to storage could also go wrong, as indicated by red steps 13 and 14, in which case the HTTP actor will return a HTTP server error.

Finally the coordinator will share with the participants it is committed (step 14) and respond with a Gone HTTP response to the client in step 15.

3.5.6 Handling Coordinator Failure While Deleting Blobs

Figure 3.8 shows the same process for deleting blobs as figure 3.7, hiding certain steps and instead showing what happens if the node acting as coordinator in the 2PC query fails. The numbering for the steps showing the successful path are kept the same as in figure

3.7, as a result the red steps create its own order in some cases. Note that this essentially the same as section 3.5.3, but with "insert" replaced with "delete".

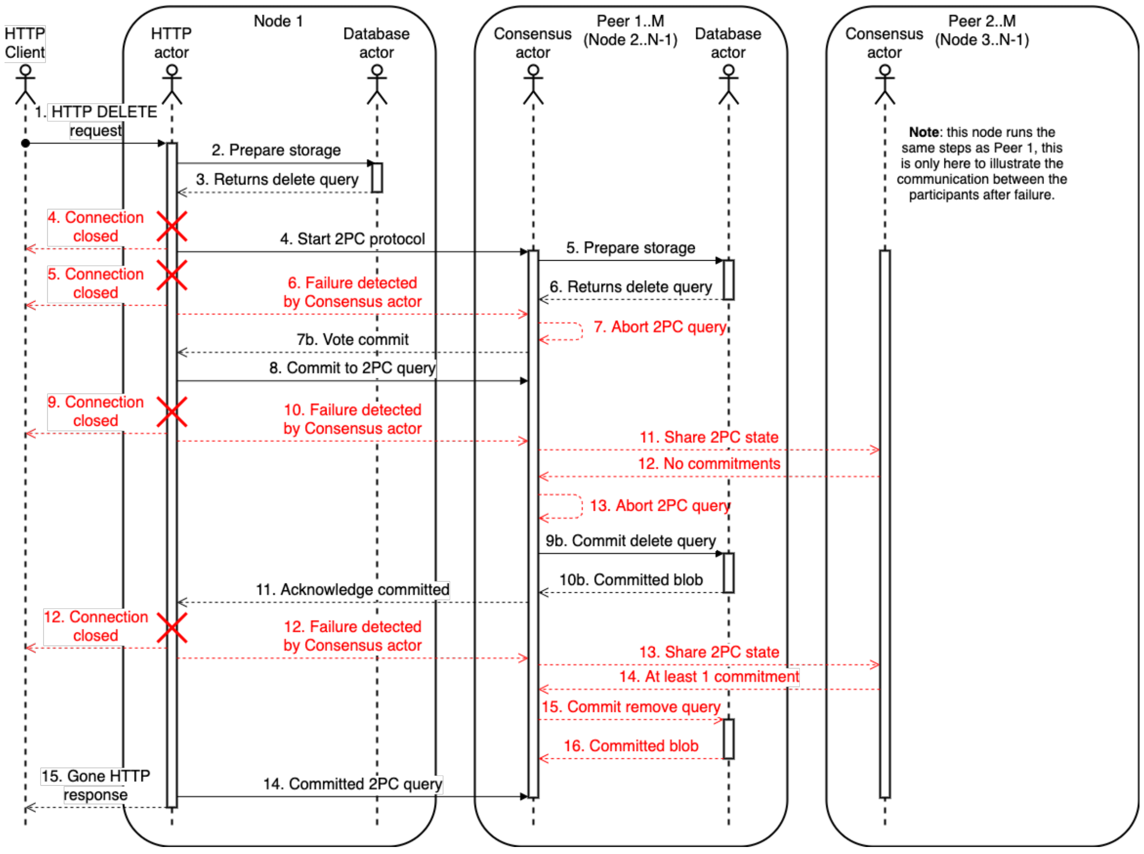


Figure 3.8: Handling coordinator failure when deleting a blob.

There are a number of cases that can occur if the coordinator node fails. If the coordinator fails before step 4 (starting the 2PC protocol) we don't have to do anything special as the 2PC algorithm hasn't started. The connection to the client will be (forcefully) closed and the client will see this as an error, this is shown in step 4 (red). The result is that the blob is not deleted.

If the coordinator fails after step 4, but before step 8 (committing of the 2PC query) in the process, the 2PC query will be aborted. The participants will detect that the coordinator has failed and will abort the 2PC query as shown in red steps 6 and 7. Like in the previous case: the client will be disconnected and will see this as an error as shown in step 5 (red). The blob is not deleted.

What happens if the coordinator crashes after committing to the 2PC query in step 8 is more complicated. It could be that some of the participants have received a commit message, but others have not. In this case we need finish the second phase of the 2PC protocol using the participants alone. Once a participant detects that the coordinator has crashed it will share this information with the all other participants, and whether or not the coordinator send a message to commit to the 2PC query (red steps 11 and 12). If no participants received a message to commit the 2PC query, it will be aborted, as shown by red step 13. From the client perspective it will be if the request failed in an earlier stage of the process, similar to the second case (failing between steps 4 and 8).

If however one or more participants committed to the query they share this with all the other participants (red steps 13 and 14), which will also commit to the query as shown by red steps 15 and 16. The result will be that the blob is deleted, but the client will not be informed (as the connection to it has already been disconnected).

3.5.7 Compacting

As described in section 3.5.5, for performance reasons the bytes that make up a blob aren't removed from the data file. This means that the data file grows infinitely, even if blobs are deleted. To prevent this infinite growth the storage on disk can be compacted. In this process all bytes that do not have an index entry pointing to them, i.e. all blobs that were never committed or blobs that were deleted, will be removed and the data file size will be reduced.

A quick version of the compacting process can be done by scanning all index entries and looking for entries that are deleted, but still have a non-zero offset (into the data file). This indicates that the bytes that make up the blob are still in place, as the deleting process only marks the index entry as deleted and doesn't touch the offset. The first step is setting the deleted blob index entry's offset to zero. Next we need to find a blob at the end of the data file that fits nicely into this gap we created (were the deleted blob was previously stored). After finding such a blob we copy it into the gap and sync it to disk, leaving the original bytes in place. If we don't find a blob that fits the existing space we leave the bytes to be compacted in the longer version (see below). We then update the copied blob index entry, changing the offset to the new location (and syncing it to disk). Finally we can remove the bytes of the moved blob, freeing up space at the end of the data file. This allows use to truncate the file safely without losing data.

A longer version of the compacting process also looks at bytes which have no index entry at all, e.g. blobs that were never committed. For the process Store*d* needs to create ranges of unused bytes into which blobs can be moved. We start with a range which covers the entire data file, i.e. starting at byte 0 and ending at the end of the file. Next we loop over all index entries: for deleted blobs we set the index's offset and length to zero, for the other entries (for stored blobs) we remove the range of bytes (that make up to blob) from the to be removed range (which covered the entire file previously). After looping over all index entries we should be left with a list of ranges with unused bytes. Depending on the size of unused bytes we can choose to move blobs, like we did in the quick version above, or we can move up all blobs rather then just filling the unused gap.

This currently is not implemented in the prototype.

3.5.8 Validating

Using the SHA-512 checksum of the blob as key not only gives us a deterministic way to create the key, but it also gives a way to check the validity of the blob itself, both in transit and on disk. Clients can use the key to validate the blob when reading and inserting the blob. Store*d* also checks the blob when communicating between peers.

Blobs stored on disk are validated using the same method. Store*d* simply iterates over the stored blobs, reading each from disk, creating the SHA-512 checksum for each and comparing the checksum to the blob's key. For all corrupted blobs their index entry will be invalided (index removed, not blob marked as deleted) and the blob will be requested from a peer when synchronising blobs on startup (see section 3.5.1).

Validating blobs on disk is not implemented in the prototype.

3.5.9 Recovery

There are two scenarios from which Store*d* needs to be able to recover: server crashes and network interruptions.

Once a crashed server is restarted it will run a full synchronisation, as described in section 3.5.1. After the startup procedure Store*d* should be fully functional again. But during the crash we could be writing to database files, for this reason the storage layer is designed and implemented to be fault tolerant, as described in 3.4.2. This relies on the assumption, as mentioned in section 3.1, that the file system functions as documented.

The process of dealing with network interruptions is similar to that of crash recovery. After two peers are reconnected (after disconnecting) the peer synchronisation[1] runs to ensure both nodes have the blobs that are inserted/deleted in the time they were disconnected from one another. However currently during network interruptions all 2PC queries will fail, meaning that blobs can not be inserted or deleted, but can be still be read.

3.5.10 Shutdown

Shutting down Store*d* starts with sending the process a signal such as `SIGTERM` or `SIGQUIT`. This signal is relayed to the HTTP and peer listener actors, shutting them both down. Existing HTTP connections, and the actors that handle them, will shutdown once the other side disconnects or after roughly two minutes once it times out. The database

[1]Peer synchronisation is the same synchronisation process as described in section 3.5.1, but only with the peer that was disconnected rather then all peers.

actor is shutdown once all references to it are dropped, which are held by the HTTP and consensus actors. Once all actors are stopped the Heph runtime stops itself.

Chapter 4

Evaluation

As previously mentioned this thesis is an experimental system research thesis, for which I made a prototype, implemented following the design described in chapter 2. This chapter shows how the prototype, Store$d$, was evaluated to answer the three research questions described in section 1.2.

RQ1 looks at the simplicity of the design of distributed data store, for which I found no standardised way to evaluate the simplicity of a design of a distributed data store. As such I can only argue it informally, which I've done in section 4.3.

The other two research questions, RQ1 and RQ2, look at the performance and scalability of a data store, for which I did find various standardised benchmarks. I decided on using the Yahoo! Cloud Serving Benchmark (YCSB) [39, 40]. YCSB is a benchmark capable of emulating different kind of key-value store workloads. Although key-value workloads are not a perfect fit for a value store, it's the closest workload I could find at the time of writing. Another well known benchmark is TPC-C [41, 42], however this focuses on on-line transaction processing (OLTP) workloads which are further away from value store workloads than key-value store workloads and thus a worse fit. The results of YCSB can be found in sections 4.4 and 4.5, which answers RQ1 and RQ2 respectively.

In the next section I'll describe the experimental setup for YCSB in more detail, as well as the stores I'll compare Store$d$ against and why.

4.1 Experimental Setup

All experiments are run on the Distributed ASCI Supercomputer 5 (DAS5) [43, 44]. Each node has a dual 8-core 2.4 GHz (Intel Haswell E5-2630-v3) CPU configuration and 64 GB memory. [1] The number of nodes used varies between benchmark runs and will be explicitly stated for each result.

[1]The nodes used are homogeneous.

Stand-alone numbers, for example latency, are rarely useful due to the use of different (hardware) setups. To put the results in perspective I'll also run the benchmarks against Redis [5] and etcd [45] to get comparable results.

One problem encountered during benchmarking was the performance of the storage on the DAS5. The default storage option (a distributed file system) supports sequential read speeds of roughly 100 MiB/s, as measured by fio [46]. That is not even close to commodity Solid State Drives (SSD), supporting read speeds well over 1 Gib/sec. For reference, a "Mid 2015" MacBook Pro has a read speed of 1500 MiB/s. The DAS5 also has a number of nodes with SSDs, but disappointingly those have read speeds of only 37 MiB/s and thus perform even worse than the distributed file system. After discussing with my thesis supervisor we decided on using in-memory file system (tmpfs [47] or ramdisk [48]) as "stable storage". However setting up such a filesystem required administrative privileges I didn't have. As a work-around I finally decided on using `/dev/shm` as stable storage. The `/dev/shm` directory is used for POSIX shared memory, essentially memory shared between processes. This directory had read speed of 800 MiB/s, which is close to the read speed of a SSD, which made the results usable.

As already mentioned I'll be using YCSB to evaluate RQ1 and RQ2. YCSB comes with six key-value workloads (A through F) and one time series workload. Of these workloads only two are used: C and D. The time series workload is an obvious bad fit and the other workloads include operations not supported by Store*d* such updating/modifying values or scanning (returning a range of values).

YCSB workload C is read-only, this would emulate the workload of a process that runs in phases with (intermediate) results being stored in Store*d*, such as machine learning data sets. Workload D's operations are 95% read and 5% insert. This is more in line with the use case of hosting static files such as videos or images, library or container registries.

YCSB works in two stages. First, the load stage in which data is loaded into the store. Second, the run stage in which the workload is benchmarked. The workload operations (i.e. the percentage read/insert) are only used in the second stage.

The Ambry paper [13] has shown that the blob size impacts performance greatly. For this reason I ran both workloads (C & D) using two blob sizes: 20 kB and 10 MB. 20 kB blobs represents for example library archives, while 10 MB represent images and other hosted media.

YCSB wasn't designed for immutable data stores and as such defines it own keys (to store a value at), but Store*d* doesn't support user-defined keys. To support these operations I created a YCSB key to Store*d* key mapping in the YCSB client, a `ConcurrentHashMap` (in Java) that maps the key provided by YCSB to the key generated by Store*d*. The overhead of this mapping is included in the reported latencies. For Redis and etcd no such client side mapping is required.

The YCSB client was run on a single DAS5 node, using 16 client threads to make requests. This means that the concurrency is 16 as well. There was no target operations per second set, leaving it unthrottled (i.e. make as many request as possible, until the target number of operations is hit).

YCSB collects the following metrics per operation (read or insert): the total number of operations, average latency, minimum latency, maximum latency, 95% and 99% latency. Note that these are client side (thus end-to-end) measurements and include a network round-trip as the YCSB client and stores are running on different nodes. In addition it keeps track of the throughput and total run time. By default YCSB includes the cleanup operations in the total run time, but I've removed it from the results below.[1] Store*d* needs to store it's YCSB key to Store*d* key mapping at the end of the load stage to be able to load the mapping again in the run stage, Redis and etcd however don't have to do anything in the cleanup stage. Because the mapping can get rather large (hundreds of megabytes in JSON format) it would be unfair to Store*d* to include this is the run time and throughput results. Note that the latencies are unchanged as the cleanup latencies are not reported in this thesis.

4.1.1 Store*d* Setup

The configuration of Store*d* is very simple and not worth discussing here. But as described in chapter 2 Store*d* creates a fully connected network and stored all blobs on all nodes.

Because the data stored by Store*d* was not needed after the benchmark run the benchmark script forcefully killed the application after the run was complete. This caused warnings to be printed when nodes detected that their peers were (uncleanly) disconnecting. These warnings did not impact the results.

4.1.2 Redis Setup

Redis is an in-memory data structure store, which supports YCSB's key-value workloads. Redis will act as a lower bound as it's in-memory, but using the OS interface (i.e. no kernel-bypass or special hardware). The evaluation is done using Redis version 6.0.8 (the latest at time of benchmarking). Store*d* and Redis vary on various points:

- Redis stores data in-memory, where Store*d* stores data on-disk. To make the comparison a little fairer I've set `appendonly` to `yes` and `appendfsync` to `always`, which forces Redis to write the request to disk followed by a call to `fsync(2)` for every write request (something Store*d* also does).

[1]This is done by removing the maximum latency for the cleanup operations from the run time, as only a single cleanup operation is run per client thread (which are run concurrently).

- Redis doesn't support synchronous replication using either Redis Cluster [49, 50] or Redis Replication [51], it only supports asynchronous replication. Store*d* on the other hand only supports synchronous replication, this will cause Redis to have an advantage when it comes to multi node performance.
- Redis Cluster partitions the values over all available instances, Store*d* however stores all blobs on all nodes. The client for Store*d* is very simple and sends all requests to a single node. This gives Redis an advantage when inserting the blobs. It will not impact the read performance as the client used by YCSB, Jedis [52], is cluster aware meaning it sends the requests to the correct node.

These differences give Redis an advantage over Store*d* (along with a decade of development work put into it), which is reflected in the results. Except for the settings mentioned above the default configuration is used. Note that the settings above, to make the comparison fairer, means that the results for Redis don't represent an "average" Redis setup, as the default configuration would likely perform better.

Redis Cluster requires a minimum of three nodes, no results were gather using two nodes for Redis.

During the benchmarking Redis logged a number of warnings. First, it warned that it was unable to enforce the TCP backlog setting of 511 because `/proc/sys/net/core/somaxconn` was set to 128. Because of the low number of connections this should not be a problem and at no point was an `ECONNREFUSED` [53] error raised (error returned when a connection was refused). Second, Redis warned that `overcommit_memory` [54] is set to `0`, saying that background saving might fail under low memory conditions. Since we're not using background saving this should not impact the results. Finally, it complained that Transparent Huge Pages (THP) were enabled, saying it would create latency and memory usage issues. Unfortunately I didn't have access to change this setting. After review I don't believe any of these warnings impacted the results of Redis.

4.1.3 etcd Setup

The default configuration for etcd was used. An initial cluster was created with all peers' URLS provided, creating a new cluster for each benchmark run. etcd and Store*d* vary on the following points:

- etcd synchronises to disk in batches and only on the coordinator node. Store*d* synchronises to disk on each insert/delete request and on all nodes.
- etcd can only store blobs of sizes up to 1.5 MiB, the limit of Store*d* is 1GB.

etcd logged two warnings during the benchmark runs. First it complained that the token was not cryptographically signed, but this should not impact the results. Second, it logged warnings once the servers were forcefully closed. Store*d* logged similar warnings and these did not impact the results as they happened after the benchmark run was complete. None of the warnings impacted the results of etcd.

4.1.4 Store Overview

Table 4.1 gives a quick overview of all benchmarked stores and some of their properties.

Property	**Store*d***	**Redis**	**etcd**
Flushes changes to disk on each write request.	Yes, flushes to disk every store request.	Yes (configured to do, see section 4.1.2).	No batches write requests.
Synchronous or asynchronous replication.	Synchronous replication to all peers.	Asynchronous replication.	Synchronous replication to a majority of the peers.
Data must fit in memory.	No.	Yes, in-memory only.	No.
Additional notes.		Acts as lower bound.	Maximum value limit is 1.5 MiB. Uses Raft [9] for consensus.

Table 4.1: Overview of the stores.

4.2 Limitations

While evaluating Store*d* I've hit a number of limitations I would like to mention. First, I attempted to benchmark more stores than reported here. Specifically Ambry, arguably the most directly-comparable stored for Store*d*, and FoundationDB. However I failed to run benchmarks against both stores, why is discussed in section 5.2.2.

Second, time. As with everything I had limited time to evaluate Store*d*. After spending too much time building Store*d* and attempting to benchmark other stores, both described in more detail in section 5.2, I couldn't run all experiments I planed for. For example I planned to do micro benchmarking to discover what the concurrent client limit of Store*d* is, to see if using a single thread to access the storage is a bottleneck. However due to the time limitation I was only able to benchmark using YCSB.

4.3 RQ1 Simpler Design

Let's start with repeating the research question: *does taking advantage of the immutable property of data lead to a simpler design of a distributed (immutable) data store?* The key factor in this question is of course *simpler*. However I couldn't find any standardised (or non standardised) way to evaluate the simplicity of a data store design. So instead of proofing Store*d* is simple, or simpler than a mutable blob store's design would be, I will present a number of arguments for the simplicity of Store*d*.

First, Store*d* is a single binary with only one kind of component. Many distributed system designs use different kinds of components (for example splitting processing and storage), making deployment more complex. Furthermore the entirety of Store*d* is implemented in three actors: HTTP/client actor (handles connections from clients), consensus actor (handles connections from peers) and the database actors (handles interaction with stable storage), as seen in figure 3.1. I would argue that, per the previous description and figure 3.1, Store*d* can be described as simple (although, again, I have no way to measure or proof it).

Second, successfully using the Two phase commit (2PC) protocol [26] for consensus, seen as one of the the simplest consensus protocols. One of the main disadvantages of 2PC is that it's a blocking protocol, only a single query can run concurrently on the data it operates on. However since the data in Store*d* is immutable multiple 2PC queries can safely be executed concurrently without interfering with one another, even on the same blob or key location. Another disadvantage of the 2PC protocol is that on coordinator failure all ongoing transactions (inserting/deleting blobs) halt, but this only the case if we have a single coordinator. However Store*d* doesn't use a single coordinator, it uses a new coordinator per transaction. This means that if a node fails it only impacts the transactions in which it plays the coordinator role.[1]

Third, deterministic key generation which doubles as checksum. Store*d* uses the SHA512 checksum [22] of the blob as the key, which is used in various places to validate the correctness of the blobs. Because of this Store*d* doesn't need to store any additional checksum data, the key can determine if the blob is correctly retrieved from disk.

In summary, I can't proof that taking advantage of the immutable property of data leads to a simpler design. However I do personally believe this is the case, as I've argued above.

[1]In the current implementation it's required that all nodes store all blobs, which means that even a single node failing as participant will cause the 2PC protocol to fail. However this can be fixed by using portioning, as discussed in section 5.4.1.

4.4 RQ2 Performance

Research question 2: *does a distributed store designed for immutable data perform better than stores designed for mutable data?* As previously mentioned I decided on using the Yahoo! Cloud Serving Benchmark (YCSB) as performance benchmark. This section only looks at single node performance, section 4.5 will look at the performance using multiple nodes. To quantify performance I've used two metrics: latency and throughput.

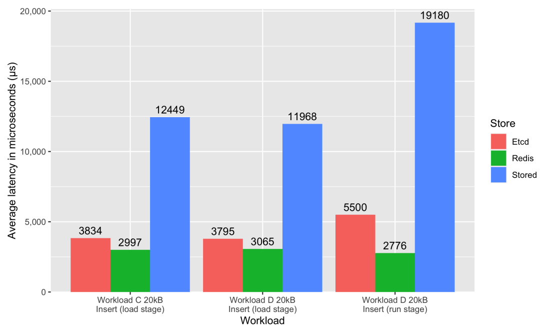
The following two subsections look at latency and throughput respectively, followed by a discussion of the results in subsection 4.4.3.

4.4.1 Single Node Latency

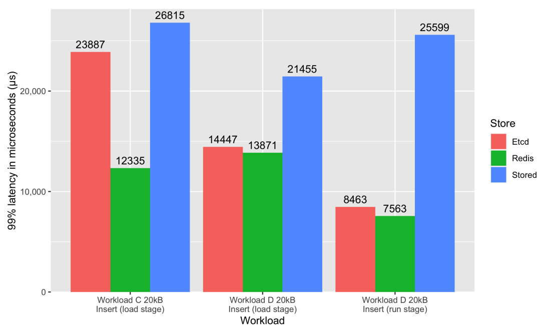
Starting with insert operations using 20 kB blobs, table 4.2 shows the average, minimum, maximum, 95% and 99% latencies. The measurements are reported in microseconds (μs), 1 millionth (1,000,000) of a second, unless stated otherwise. Figure 4.1 shows the average and 99% latencies for inserting 20 kB blobs, these are the same as in table 4.2.

Store	**Avg. latency**	**Min. latency**	**Max. latency**	**95% Latency**	**99% latency**
			Workload C load stage		
Store*d*	12,449 μs	671 μs	1,935,359 μs	19,711 μs	26,815 μs
Redis	2,997 μs	776 μs	69,759 μs	4,635 μs	12,335 μs
etcd	3,834 μs	533 μs	1,649,663 μs	2,249 μs	23,887 μs
			Workload D load stage		
Store*d*	11,968 μs	611 μs	1,808,383 μs	17,343 μs	21,455 μs
Redis	3,065 μs	648 μs	56,191 μs	4,187 μs	13,871 μs
etcd	3,795 μs	544 μs	1,657,855 μs	2,369 μs	14,447 μs
			Workload D run stage		
Store*d*	19,180 μs	933 μs	1,949,695 μs	16,263 μs	25,599 μs
Redis	2,776 μs	1,024 μs	13,327 μs	5,723 μs	7,563 μs
etcd	5,500 μs	838 μs	430,079 μs	6,795 μs	8,463 μs

Table 4.2: Insert latencies using 20 kB blobs.



(a) Average latency.



(b) 99% latency.

Figure 4.1: Insert latencies using 20 kB blobs.

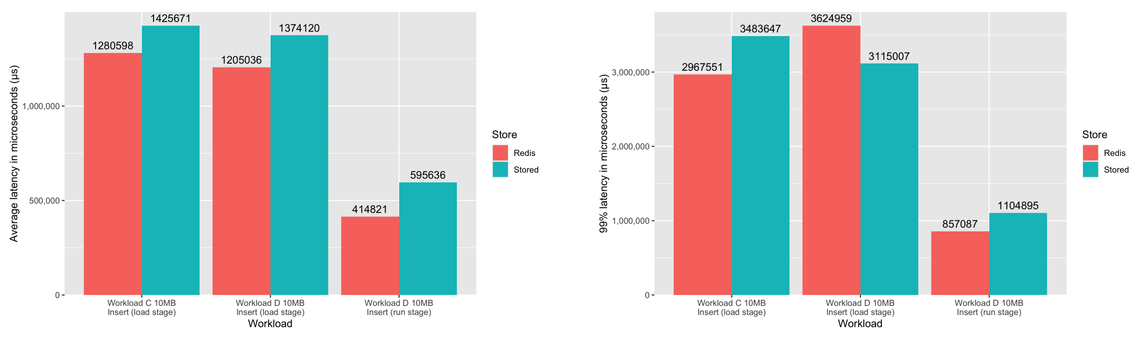
The table and figure show a clear picture with regards to 20 kB blobs: Redis and etcd have a lower latency than Store$d$. Store$d$ has a 2-7x higher average and 99% latencies, a difference of 10-20 milliseconds. Store$d$ performs significantly worse on maximum latency. The maximum latency is nearly 2 seconds for all three workloads, which is 30-150x higher than Redis.

Next we'll look at a similar table and figure as above, but for the latencies of inserting 10 MB blobs. Note that etcd doesn't support blobs larger than 1.5 MiB and thus isn't included in these results. Table 4.3 shows all the latencies exported by YCSB, figure 4.2 shows the average and 99% latencies.

Store	**Avg. latency**	**Min. latency**	**Max. latency**	**95% Latency**	**99% latency**
Workload C load stage					
Store$d$	1,425,671 μs	368,896 μs	5,578,751 μs	2,248,703 μs	3,483,647 μs
Redis	1,280,599 μs	170,112 μs	4,366,335 μs	2,263,039 μs	2,967,551 μs
Workload D load stage					
Store$d$	1,374,120 μs	337,664 μs	4,263,935 μs	2,553,855 μs	3,115,007 μs
Redis	1,205,036 μs	251,264 μs	4,833,279 μs	2,359,295 μs	3,624,959 μs
Workload D run stage					
Store$d$	595,636 μs	386,816 μs	1,174,527 μs	968,191 μs	1,104,895 μs
Redis	414,821 μs	206,848 μs	857,087 μs	826,879 μs	857,087 μs

Table 4.3: Insert latencies using 10 MB blobs.

Store$d$ performs better with 10 MB blobs than it does with 20 kB blobs relative to Redis. Where Store$d$ was about 2-7 times slower than Redis using 20 kB blobs, this is reduced to getting within 40% of Redis's average, 95% and 99% latencies using 10 MB blobs. Store$d$ has a 1.3-2.2x higher minimum latency than Redis. The maximum latency for workload C



(a) Average latency. **(b)** 99% latency.

Figure 4.2: Insert latencies using 10 MB blobs.

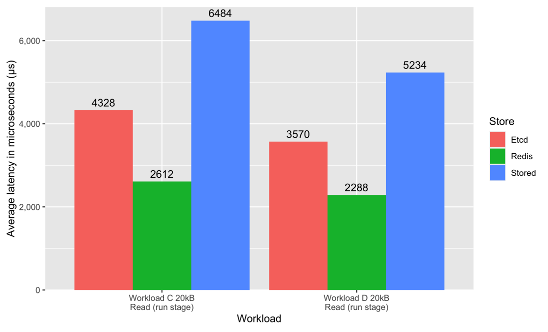
is also a second higher, but for the load stage of workload D Store*d*'s maximum latency is lower than Redis.

Next, we'll look at read operations. Following the same structure as for the insert operations, table 4.4 shows all the latencies for the read operations using 20 kB blobs, figure 4.3 shows the average and 99% latencies.

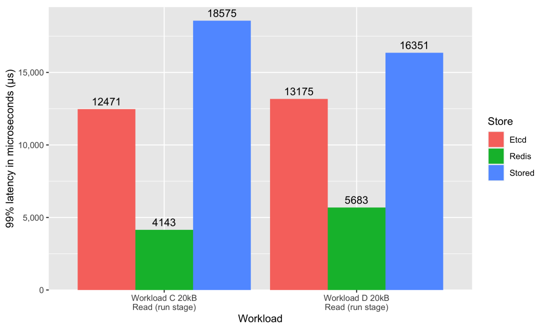
Store	**Avg. latency**	**Min. latency**	**Max. latency**	**95% Latency**	**99% latency**
			Workload C run stage		
Store*d*	6,484 μs	391 μs	1,959,935 μs	3,463 μs	18,575 μs
Redis	2,612 μs	527 μs	36,543 μs	3,757 μs	4,143 μs
etcd	4,328 μs	525 μs	732,671 μs	4,415 μs	12,471 μs
			Workload D run stage		
Store*d*	5,234 μs	472 μs	1,999,871 μs	12,559 μs	16,351 μs
Redis	2,288 μs	456 μs	36,063 μs	5,035 μs	5,683 μs
etcd	3,570 μs	521 μs	509,695 μs	4,091 μs	13,175 μs

Table 4.4: Read latencies using 20 kB blobs.

Store*d* achieves the lowest minimum latency in workload C, but also has the highest maximum latency in both workloads. The average, 95% and 99% latencies of Store*d* are close to that of Redis and etcd, Store*d* has 0.9-4.5x higher latencies compared to Redis and 0.7-3.1x higher than etcd.



(a) Average latency.



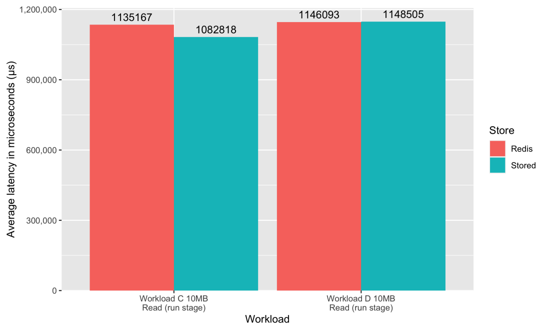
(b) 99% latency.

Figure 4.3: Read latencies using 20 kB blobs.

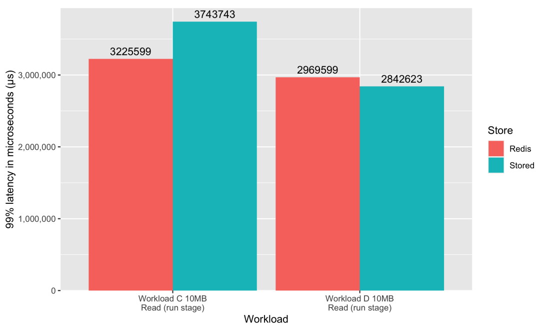
Moving on to read operations using 10 MB blobs, table 4.5 shows the all latencies, figure 4.4 shows the average and 99% latencies.

Store	**Avg. latency**	**Min. latency**	**Max. latency**	**95% Latency**	**99% latency**
			Workload C run stage		
Store*d*	1,082,818 μs	88,064 μs	4,345,855 μs	2,897,919 μs	3,743,743 μs
Redis	1,135,167 μs	104,256 μs	3,909,631 μs	2,525,183 μs	3,225,599 μs
			Workload D run stage		
Store*d*	1,148,505 μs	88,128 μs	3,788,799 μs	2,273,279 μs	2,842,623 μs
Redis	1,146,093 μs	108,160 μs	3,747,839 μs	1,825,791 μs	2,969,599 μs

Table 4.5: Read latencies using 10 MB blobs.



(a) Average latency.



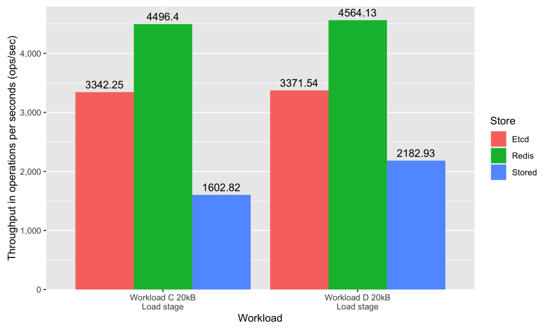
(b) 99% latency.

Figure 4.4: Read latencies using 10 MB blobs.

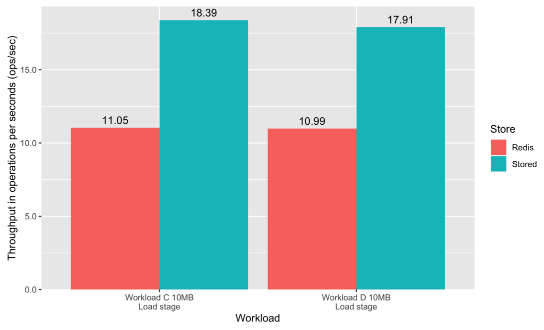
Again Store*d* performs better with 10 MB blobs than 20 kB blobs, relative to Redis. Store*d* achieves better results than Redis in some categories, e.g. the minimum latency of workload D is 19% lower than that of Redis. In the cases where Store*d* has higher latencies they are 1.01-1.25x higher than the latencies of Redis.

4.4.2 Single Node Throughput

To answer the research question I needed quantify performance, for I've used two metrics: latency and throughput. Now we got the results for latency, we'll look at throughput next. Starting with insert operations in the load stage figure 4.5 shows the throughput using 20 kB and 10 MB blobs.



(a) 20 kB blobs.

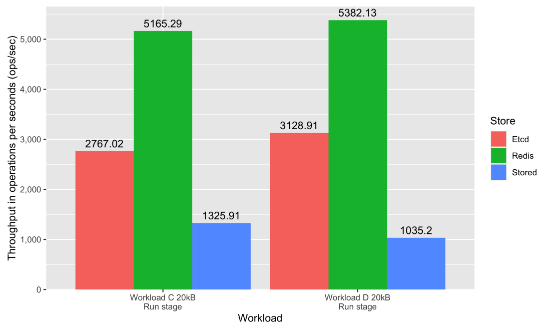


(b) 10 MB blobs.

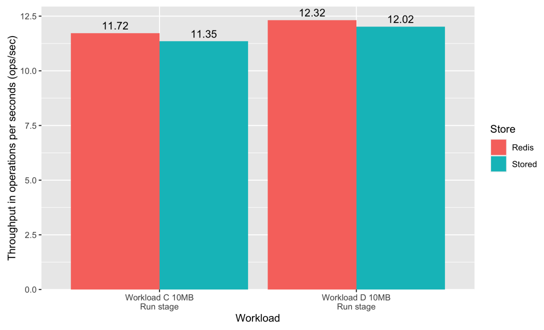
Figure 4.5: Total throughput during load stage.

Redis has a throughput 2.1-2.8x higher than that of Store$d$ inserting 20 kB blobs, however Store$d$ has a 1.6x higher throughput when using 10 MB blobs. etcd performs between Redis and Store$d$, getting 2x higher throughput than Store$d$.

Next we'll look at the read operations. Same setup as previously, figure 4.6 shows the throughput for 20 kB blobs and 10 MB blobs.



(a) 20 kB blobs.



(b) 10 MB blobs.

Figure 4.6: Total throughput during run stage.

Redis achieves 3.9-5.2x higher throughput with 20 kB blobs than Store$d$, but the throughput when using 10 MB blobs is within 3% of that of Store$d$. etcd has 2.1-3x higher throughput using 20 kB blobs than Store$d$.

4.4.3 Discussion

To summarise the results, Redis and ectd both achieve lower insert and read latencies than Store*d* in most cases. Redis and etcd also have a higher throughput when using 20 kB blobs than Store*d*, however when using 10 MB blobs Store*d* has equal or better throughput than Redis (etcd doesn't support 10 MB blobs).

The insert latency can be explained by the amount of `write(2)` and `fsync(2)` (equivalent)[1] system calls made. Redis uses a single `write` and `fsync` call to write the request to disk, however Store*d* uses both calls twice, for the data file and for the index file. etcd batches write requests when synchronising them to disk, compared to Store*d* which synchronises on every write.

For the read operations I don't have an explanation why Redis or etcd is faster, however I do have an idea to make Store*d* faster which is discussed in section 5.4.2.

Lastly I want to note that I believe the high maximum latency of Store*d* can be attributed to the poor scheduler in Heph (the framework Store*d* is based on), as Store*d* is capable of minimum latencies as low or lower than both Redis and etcd in various cases.

Considering both Redis and etcd are multi-year projects with many contributors and Store*d* is only created by myself in roughly 4 months, I think there is more potential in Store*d*, some of which is discussed in section 5.4.

Finally to answer the research question: *does a distributed store designed for immutable data perform better than stores designed for mutable data?* For Store*d*, compared to Redis and etcd, the answer is *no*. However I think that the results show a lot of promise when it comes to using the immutability aspect of data.

4.5 RQ3 Scalability

The final research question is the following: *does a distributed store designed for immutable data scale better than stores designed for mutable data?* It looks at the performance of the store from a scalability perspective. To answer this question I again used YCSB and report the same metrics as done in section 4.4. Only a single YCSB client was run that send its requests to a single node. As previously mentioned Redis Cluster requires at least three nodes to run, so no results were gathered using two nodes for Redis.

This section has the same structure as the previous section, first looking at the latency and throughput results, followed by a discussion of the results in subsection 4.5.3.

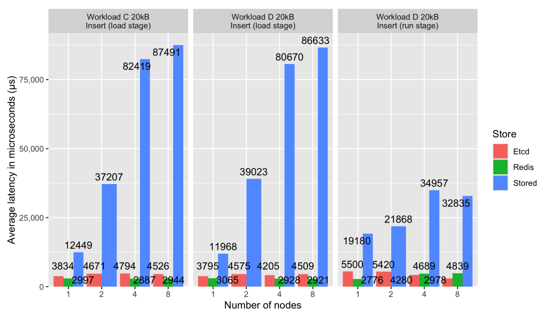
[1]Store*d* uses `msync(2)` instead of `fsync(2)` to synchronise the data file to disk, but both system calls achieve the same thing.

4.5.1 Multiple Nodes Latency

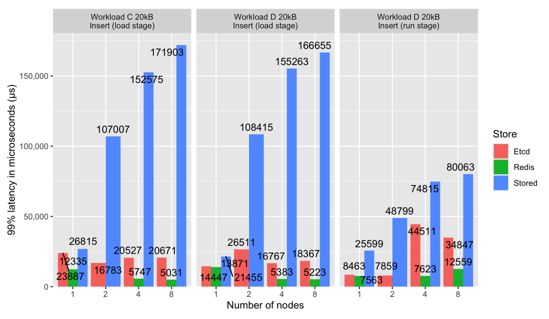
Table 4.6 and figure 4.7 show the average and 99% latencies inserting 20 kB blobs using multiple nodes.

Store	1 Node	2 Nodes	4 Nodes	8 Nodes	1 Node	2 Nodes	4 Nodes	8 Nodes
Workload C load stage								
	Average latencies				99% latencies			
Store*d*	12,449 μs	37,207 μs	82,419 μs	87,491 μs	26,815 μs	107,007 μs	152,575 μs	171,903 μs
Redis	2,997 μs	N/A	2,887 μs	2,944 μs	12,335 μs	N/A	5,747 μs	5,031 μs
etcd	3,834 μs	4,671 μs	4,794 μs	4,526 μs	23,887 μs	16,783 μs	20,527 μs	20,671 μs
Workload D load stage								
	Average latencies				99% latencies			
Store*d*	11,968 μs	39,023 μs	80,670 μs	86,633 μs	21,455 μs	108,415 μs	155,263 μs	166,655 μs
Redis	3,065 μs	N/A	2,928 μs	2,921 μs	13,871 μs	N/A	5,383 μs	5,223 μs
etcd	3,795 μs	4,575 μs	4,205 μs	4,509 μs	14,447 μs	26,511 μs	16,767 μs	18,367 μs
Workload D run stage								
	Average latencies				99% latencies			
Store*d*	19,180 μs	21,868 μs	34,957 μs	32,835 μs	25,599 μs	48,799 μs	74,815 μs	80,063 μs
Redis	2,776 μs	N/A	4,689 μs	4,839 μs	7,563 μs	N/A	7,623 μs	12,559 μs
etcd	5,500 μs	5,420 μs	4,280 μs	2,978 μs	8,463 μs	7,859 μs	44,511 μs	34,847 μs

Table 4.6: Insert latencies using 20 kB blobs.



(a) Average latencies.



(b) 99% latencies.

Figure 4.7: Insert latencies using 20 kB blobs.

Store*d* has 3-30x higher average latency than both Redis and etcd. Furthermore with each added node Store*d* has a higher latency, only the increase from four to eight nodes doesn't see a nearly linear increase in latency. Redis and etcd however both maintain roughly the same average latency independent of the number of nodes.

Store*d* also sees a nearly linear increase in 99% latency when using one, two or four nodes, same as with the average latency. Store*d* has a 1.1-34x higher 99% latency than Redis and etcd. The 99% latencies of Redis and etcd fluctuate more than their average latencies.

Now we'll look at 10MB blobs. Table 4.7 and figure 4.8 show the average and 99% insert latencies with 10 MB blobs. *Note that the times in table 4.7 are in milliseconds (ms) not microseconds as seen previously*,[1] figure 4.8 uses microseconds. Also note that etcd can't handle values larger than 1.5 MiB and thus isn't included in these results.

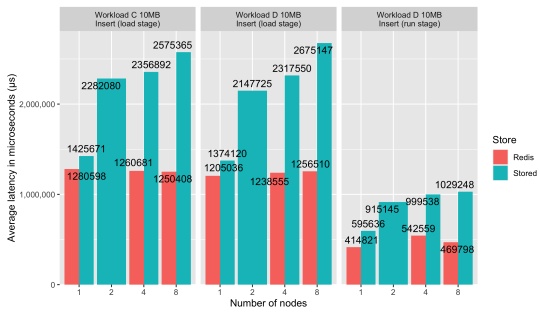
Store	**1 Node**	**2 Nodes**	**4 Nodes**	**8 Nodes**	**1 Node**	**2 Nodes**	**4 Nodes**	**8 Nodes**
Workload C load stage								
	Average latencies				99% latencies			
Store*d*	1,426 ms	2,282 ms	2,3567 ms	2,575 ms	3,484 ms	4,260 ms	4,735 ms	4,542 ms
Redis	1,281 ms	N/A	1,261 ms	1,250 ms	2,968 ms	N/A	3,715 ms	3,686 ms
Workload D load stage								
	Average latencies				99% latencies			
Store*d*	1,374 ms	2,148 ms	2,318 ms	2,675 ms	3,115 ms	4,096 ms	3,918 ms	5,243 ms
Redis	1,205 ms	N/A	1,239 ms	1,257 ms	3,625 ms	N/A	3,701 ms	4,192 ms
Workload D run stage								
	Average latencies				99% latencies			
Store*d*	596 ms	915 ms	1,000 ms	1,029 ms	1,105 ms	1,831 ms	1,478 ms	1,798 ms
Redis	415 ms	N/A	543 ms	470 ms	857 ms	N/A	2,476 ms	985 ms

Table 4.7: Insert latencies using 10 MB blobs.

Compared to results from the 20 kB blobs Store*d* does better, but the average latency is still 1.1-2.1x higher than that of Redis. Furthermore Store*d* still sees an increase in average latency with each added node.

Store*d* achieves a better 99% latency than Redis in the load stage of workload D, but in all other categories Store*d* has a 1.2-2.1x higher 99% latencies than Redis.

[1]Because the table wouldn't fit on the page reporting the measurements in microseconds.


(a) Average latencies.

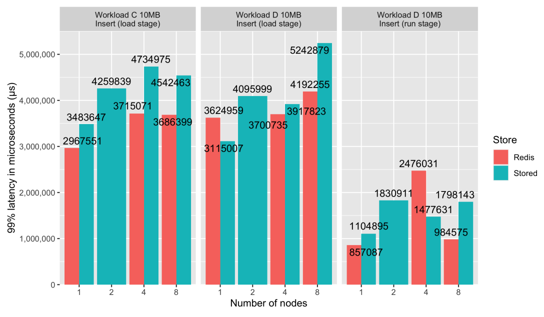
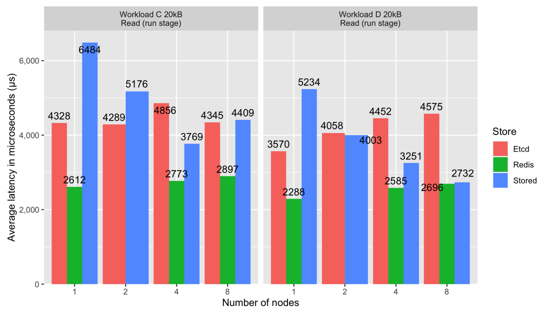

(b) 99% latencies.

Figure 4.8: Insert latencies using 10 MB blobs.

Those were the results for the insert operations. Next we'll look at the read operations. Table 4.8 and figure 4.9 show the average and 99% read latencies using 20 kB blobs.

Store	1 Node	2 Nodes	4 Nodes	8 Nodes	1 Node	2 Nodes	4 Nodes	8 Nodes
				Workload C				
		Average latencies				99% latencies		
Stored	6,484 μs	5,176 μs	3,769 μs	4,409 μs	18,575 μs	22,719 μs	21,695 μs	29,391 μs
Redis	2,612 μs	N/A	2,773 μs	2,897 μs	4,143 μs	N/A	4,951 μs	6,787 μs
etcd	4,328 μs	4,289 μs	4,855 μs	4,345 μs	12,471 μs	19,359 μs	12,159 μs	10,895 μs
				Workload D				
		Average latencies				99% latencies		
Stored	5,234 μs	4,003 μs	3,251 μs	2,732 μs	16,351 μs	18,943 μs	20,031 μs	13,151 μs
Redis	2,288 μs	N/A	2,585 μs	2,696 μs	5,683 μs	N/A	4,827 μs	5,463 μs
etcd	3,570 μs	4,058 μs	4,452 μs	4,575 μs	13,175 μs	53,407 μs	19,359 μs	9,215 μs

Table 4.8: Read latencies using 20 kB blobs.


(a) Average latencies.

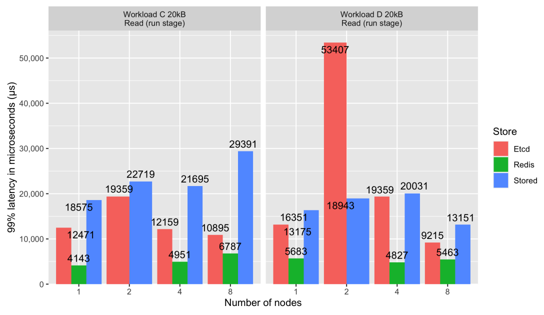

(b) 99% latencies.

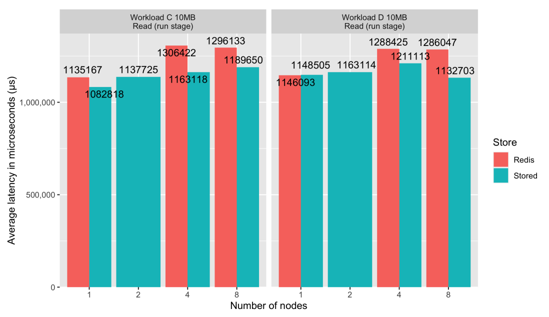
Figure 4.9: Read latencies using 20 kB blobs.

Unlike with insert operations Stored doesn't need to interact with its peers for reads, so we don't see the same increase in latency when increasing the number of nodes as we did for the inserts. However Stored still has the highest latencies in most cases, which are 1-2.5x times higher than Redis and 0.6-1.5x higher than etcd's latencies.

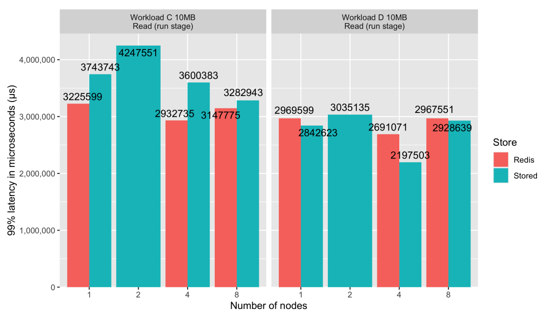
Next are the 10 MB blobs. Table 4.9 and figure 4.10 show the average and 99% read latencies using 10 MB blobs. *Note the measurements in table 4.9 are reported in milliseconds, not microseconds*, figure 4.10 uses microseconds.

Store	**1 Node**	**2 Nodes**	**4 Nodes**	**8 Nodes**	**1 Node**	**2 Nodes**	**4 Nodes**	**8 Nodes**
Workload C								
Average latencies					99% latencies			
Stored	1,083 ms	1,138 ms	1,163 ms	1,190 ms	3,744 ms	4,248 ms	3,600 ms	3,283 ms
Redis	1,135 ms	N/A	1,306 ms	1,296 ms	3,226 ms	N/A	2,933 ms	3,148 ms
Workload D								
Average latencies					99% latencies			
Stored	1,149 ms	1,163 ms	1,211 ms	1,133 ms	2,843 ms	3,035 ms	2,198 ms	2,929 ms
Redis	1,146 ms	N/A	1,288 ms	1,286 ms	2,970 ms	N/A	2,691 ms	2,968 ms

Table 4.9: Read latencies using 10 MB blobs.



(a) Average latencies.



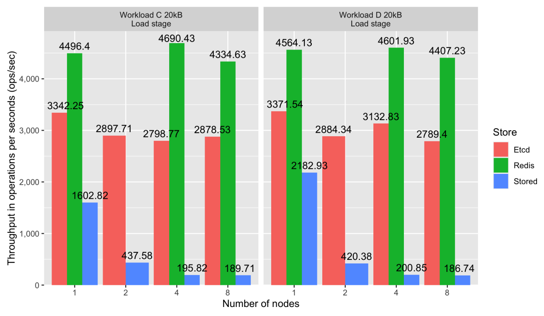
(b) 99% latencies.

Figure 4.10: Read latencies using 10 MB blobs.

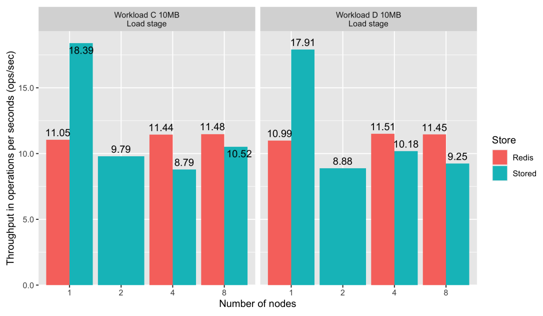
As we've seen in the single node results Stored is much closer to Redis's performance using 10 MB blobs. The average latency of Stored is better in five cases (up to 14%) and worse (by 1%) in a single case compared to that of Redis. The 99% latency of Stored is worse than that of Redis using workload C (up to 23%), but using workload D Stored has lower latencies (also up to 23%).

4.5.2 Multiple Nodes Throughput

The final metric we'll looking at to answer the research question is throughput. Figure 4.11 shows the throughput during the load stage (thus at insert operations) for blobs of 20 kB and 10 MB.



(a) 20 kB blobs.

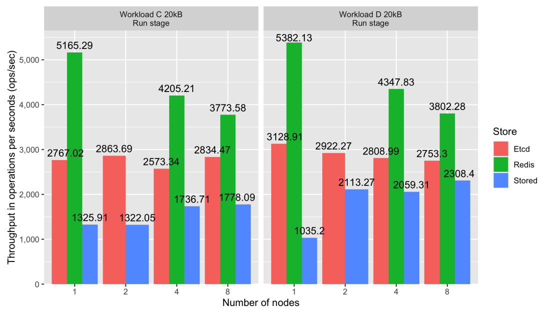


(b) 10 MB blobs.

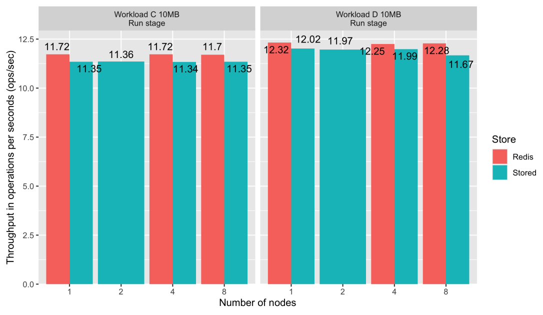
Figure 4.11: Total throughput during load stage.

For 20 kB blobs the results are clear, Redis achieves a 2.1-23.9x higher throughput than Store*d*, etcd 1.5-15.6x higher. Using 10 MB blobs however Store*d* achieves a higher throughput using a single node in both workloads. However when using multiple nodes Redis achieves a 1.1-1.3x higher throughput than Store*d*.

Next let's look at the reads during the run stage of the benchmark. Figure 4.12 looks at the throughput using 20 kB Blobs and 10 MB blobs during the run stage of the workload.



(a) 20 kB blobs.



(b) 10 MB blobs.

Figure 4.12: Total throughput during run stage.

Using 20 kB blobs Store*d* again has the lowest throughput, Redis and etcd being 1.6-5.2x and 1.2-3x higher respectively. However using 10 MB blob Store*d* is capable of keeping up with Redis this time achieving a throughput of within 5% of that of Redis.

4.5.3 Discussion

Using multiple nodes we saw similar results to those of the single node performance. To summarise the results, Redis and ectd both achieve lower insert and read latencies than Store*d* in most cases. Store*d* does better with 10 MB blobs than 20 kB blobs, but doesn't have conclusively lower latencies than Redis using 10 MB blobs (etcd doesn't support 10 MB blobs). Redis and etcd also have an order of magnitude higher throughput when using 20 kB blobs than Store*d*. Store*d* again does better with 10 MB blobs, but Redis still has a higher throughput (up to 30%) when using multiple nodes.

Redis's low insert latencies can be explained by the use of asynchronous replication, compared to Store*d* use of synchronous replication. This explains why the insert latencies remain (more or less) consistent for Redis, not increasing when adding more nodes. Furthermore the Redis client used is cluster aware, which means it knows to what node to send a certain request, splitting the read requests among the nodes. For Store*d* all requests, read and insert, are send to a single node.

Like Store*d*, etcd also uses synchronous replication, but only does this to a majority of its peers. It batches multiple insert requests when synchronising to disk and only does so on the coordinator node (which is dynamic in etcd), where Store*d* does this on each insert on all nodes.

Answering the research question: *does a distributed store designed for immutable data scale better than stores designed for mutable data?* No. When it comes to scalability it's more important to reduce the communication required between peers to keep them in a consistent state, at least when it comes to this work immutability did not help in this area.

Finally to answer the main research question: *does taking advantage of the immutable property of data lead to a better distributed (immutable) data store?* As two of the three answers to the sub-questions are no, the conclusion is simple: no. Store*d* does not sufficiently show that it's *better* than any of the mutable data stores, due to the fact it's an immutable data store.

Chapter 5

Discussion & Future Work

In this chapter I want to share some more thoughts about Store*d*, my experience working on this thesis and future work in immutability and for Store*d*.

A number of stores, including f4, Ambry and Redis, use asynchronous replication, usually for performance reasons. For Store*d* I decided against this and use synchronous replication instead, arguing that stores should be strongly consistent in section 2.2. However after the evaluation I'm more open to asynchronous replication and providing strong consistency at the proxy or client level, not at store node level. This would require a proxy or client that is aware of the distributed nature of Store*d* and any potential partitioning of the keys (see section 5.4.1 below). The proxy or client would need to ensure that requests for a certain key are send to the same store node to still provide a consistent view to the clients of Store*d*, while the store nodes themselves are eventually consistent. But this still leaves a "gap" in time between when the client receives the response to an insert request and the time the blob is properly replicated to multiple nodes.

5.1 Experience

Since this is the first distributed system I've designed and built myself I wanted to share some of my experiences. You could see this as an experience report to current and future master students who are also considering building a distributed system (spoiler alert: *it's hard*). Note that the writing in these sections is more informal than the other sections of the thesis.

First I'll say that although it's quite challenging to design a system that will operate correctly (for some definition of correctly) under all circumstances (you can think of and the ones you can't think of), it is also very fun. In the end I couldn't make Store*d* peer fault tolerance, as described in section 2.4, but I made that decision as a clear trade-off between spending even more time on the prototype and finishing my thesis.

Second, 2-3 months, or even the roughly 5 months I eventually took, isn't enough time to build a distributed system by yourself from scratch, even a relatively simple one like Store*d*. For future master student I highly recommend to *not* start from scratch. Use the available libraries (that weren't available to me) for things like task management, consensus, etc. Building, testing and debugging these kinds of complex components take time, time you don't have while working on your thesis. Instead use that time experimenting.

Instead of building something from scratch I would modify an existing system to test your hypothesis(es). The reduces the amount of work required (and perhaps the fun), but it allows you to focus on the topic at hand.

5.1.1 Using Rust

For Store*d* I've the Rust programming language. The language itself is quite young, something that is reflected in the number of crates (Rust terminology for libraries) that are not quite ready for production yet and the crates that are simply missing. I've working with Rust for about four years (since 2015) before starting the thesis and two years on Heph. After the steep learning curve, which made me quit two attempts prior, everything started to make sense. The Rust compiler wants you do things a certain way, but that way is usually better than the one you used before (with a few exceptions). I believe learning and using Rust actually made me a better all-round programmer. For example Rust's concept of *lifetime*s makes me document the valid use of pointers in my C code, explaining the "lifetime" of allocations in comments rather than code (like you would in Rust).

Right now I would recommend using Rust for building new systems, distributed or otherwise. But I would like to note that the ecosystem is still young and somewhat divided.[1] For example there are no de facto frameworks/libraries to use for certain tasks and most libraries that are available are not yet at a version 1, meaning the API is still unstable and can change. However in the five years I've used Rust things are moving into the right direction and moving fast.

5.2 Encountered Problems

There also less fun part to designing/building a distributed system. When something goes wrong in a "normal" system (i.e. not distributed) you can use conventional tools, e.g. `gdb` or `lldb` (or good ol' `print` statements), to debug the problem. This becomes much harder when it comes to distributed systems, especially if some kind of timing and communication between the nodes is involved. So I want to discuss some of the problems that I've experienced that (at some point) made me doubt my sanity and everything I knew about computers.

[1]I'm not helping the problem by creating Heph, I know that.

The first problem was the mysteriousness case of the "unexpected end of file" error. For peer interaction Store*d* uses TCP, for which a connection is setup between each peer (as described in section 3.5.1). In this connection setup we ask the peer we're connecting to for all the nodes it knows to create the fully connected network. However at any point any node can fail, so I added timeouts to detect this. Then I started seeing errors saying "unexpected end of file" while reading and the other side of the connection would report timeouts. *Spend some time thinking about what the problem could be, I'll give a hint: what you're thinking about probably wasn't the problem.* After a couple of weeks of on/off debugging this I finally found the problem and it had nothing to do with the connections themselves, but with scheduling.

Each peer connection is handled by a single thread-safe Heph actor, that is run on a worker thread, but is managed by a (different) coordinator thread. The coordinator thread determines when a thread-safe actor is ready run, e.g. when getting a readiness event from `epoll(2)`/`kqueue(2)`, and schedules the thread-safe actors in a shared scheduler data structure. However the coordinator thread doesn't run any of the actors, that is the job of the worker threads. The problem was that a worker thread setup everything for the actor to run, including registering it's accepted TCP socket with `epoll(2)`/`kqueue(2)`, and *after* that adds the actor to the scheduler structure. This created the following timeline.

1. Worker thread registered the TCP socket with `epoll(2)`/`kqueue(2)` to receive readiness events (e.g. when it can be read).

2. Coordinator thread received a readiness event for the actor. It tries to mark the actor as ready to run, but can't find the actor and ignores the event as spurious.

3. Worker thread adds the actor to the scheduler structure as *inactive*, expecting the coordinator thread to mark it as active later once it's ready to run.

4. Time goes on, but the actor is not run. Eventually the other side of the connection hits a timeout and disconnects.

5. The peer disconnecting causes the coordinator to receive another readiness event (to indicate the connection is closed) and schedules the actor.

6. The actor is now run for the first time and it tries to read from the connection. However the other side is already disconnected and it reads zero bytes, returning an "unexpected end of file" error.

The problem is in point 2, the coordinator received an event too quickly and tried to mark the actor as ready to run before it was added to the scheduler structure. This problem is made worse because Heph uses edge-triggers [55], rather than level-triggers, which means a single readiness event is created when e.g. new data is ready to read until all data is read, rather than each time new data is ready (send by the other side of the connection). But the main problem was that coordinator simply discarded the event. It did this because spurious events where possible and not uncommon. However this caused some unlucky peer actors to simply not be run until the other side already closed the connection. The solution to this was simple: don't ignore the event. Now, instead of ignoring the event, a marker is placed in the scheduler structure that indicates the actor is marked as ready to run. Once a worker thread tries to add or add back (after running) a thread-safe actor to the scheduler it will see this marker and put it in the ready-to-run queue instead of the inactive list of the scheduler.

5.2.1 DAS5 Storage Performance

A second problem I've encountered was with the DAS5. I've already mentioned this in section 4.1, but the storage performance of the DAS5 is bad compared to commodity SSDs. Initially I though this would only cause bad performance results, however with long(er) running benchmarks this also starting causing timeouts in peer and database actor interaction. I tried to improve the performance by delaying the synchronisation of `mmap(2)`-ed blobs as much as possible, but it was not enough. Eventually I resorted to using an in-memory filesystem to get usable results.

5.2.2 Benchmarking Other Stores

Finally I also had a large number of problems with benchmarking Ambry and FoundationDB.

As Ambry was the most directly comparable open-source store it made sense to compare it to Store*d*. However after nearly two full weeks of trying to configure and run the benchmarks I gave up. Using a single Ambry node was possible, but not specifying a full configuration lead to many `null` pointer exceptions with unclear error messages. After spending too much time fixing them I finally managed to run YCSB using a single node. However when trying to add more nodes it kept sending metadata requests every milliseconds to all nodes. This created a large amount of traffic between nodes and used CPU cycles. I was unable to undercover and fix the cause of this in two weeks time and decided my time would be best spend elsewhere.

After the trouble I went through with Ambry I decided to timebox the setup of FoundationDB. After two days trying configurations I failed to get it run properly. Again I had to make the decision to spend my time elsewhere.

5.3 Future Work

There is a lot of future work to be done in Store*d*, that is discussed in the next section 5.4. This section will describe general future work with regards to immutable data stores.

The results of this thesis are disappointing as I was not able to confirm my hypothesis that immutable data store could be *better*[1] than mutable stores. A similar hypothesis should be tested again with a more mature immutable data store to either confirm or deny that immutability brings some of the advantages I theorised in section 1.2.

A second avenue of future work is using immutability in the consensus algorithm. We've seen from the evaluation of RQ3 (scalability) that communication between peers (required by the consensus algorithm) plays a large role in the latency and throughput. However all consensus algorithms (I know of) are designed for mutable stores. Designing a new, or adopting an existing, consensus algorithm for immutability could potentially lead to less communication and lower latency, while maintaining strong consistency.

A third avenue is making more use of non-commodity hardware (capabilities). For example [56] and [57] use Remote Direct Memory Access (RDMA) to achieve single digit microsecond latencies in their key-value stores. Immutability could make the management of the memory areas that are remotely accessible easier.

5.4 Future Work Store*d*

There are quite a lot of improvements that can be made to Store*d*, this section discusses some of them. Starting with improving on the limitations in the design, discussed in section 2.4.

5.4.1 Partitioning

For this thesis I decided that all nodes needed to store all blobs for simplicity. However moving beyond a prototype this limitation becomes a larger problem, requiring too much storage and not providing any peer fault tolerance.

A common solution to this limitation is partitioning [58], where only a portion of the nodes (enough to achieve the desired fault tolerance) store a blob. This technique is used by many stores, including Redis, etcd, f4 and Ambry. Using partitions would solve the first limitations, that the storage requirement increases per node added, by limiting the amount of nodes that store a blob.

The second limitation, that a single failing node blocks all 2PC queries, would be improved as only a portion of the nodes would have to participate in the 2PC query, rather than all nodes. However it wouldn't solve the limitation completely as a failing node can

[1]See section 1.2 for a definition of better.

still block all 2PC queries in which that node needs to participate (based on the partitioning).

To solve the second limitation completely rather than using a fixed partitioning scheme, Store*d* would need to use a dynamic partitioning that can be adjusted based on the available/reachable nodes while running, while also maintaining Store*d* strong consistency guarantees. This is not an easy problem to solve and I don't have a recommendation of such a partitioning algorithm at the time of writing.

5.4.2 Optimise Reading Blobs

In the current implementation of Store*d* all datastore accesses, i.e. inserting, deleting and reading, have to go through the *database actor*. This has a number of benefits as discussed previously, but with an increasing number of concurrent requests this could become a bottleneck.[1] A possible optimisation here is optimising for reading blobs, which makes sense for use cases which see >95% read traffic (see section 1.3).

Another database project in Rust, Noria [36], uses an left-right [59] concurrent `HashMap` that allows reading from multiple threads without using any locks/synchronisation and a single writer. This implementation is provided as a library (`evmap` [60]) and has benchmarks that show linear read performance when adding readers.

Using a left-right concurrent data structure, such as the one provided by `evmap`, I expect the read latencies to improve by avoiding a RPC with the database actor. Furthermore with many concurrent clients issuing a mix of read/write requests I expect the insert/delete latencies to remaining lower by alleviating the message pressure on the database actor.

5.4.3 Heph Improvements

The last improvements to Store*d* isn't actually to Store*d*, but to Heph the framework I've built Store*d* on. Although the core design of Heph focuses on performance, the actual implementation has many areas of possible improvement. Based on the high maximum and 99% latencies, often measured in seconds not micro/milliseconds, and the low minimum latencies, showing that Store*d* can in fact achieve low latencies, from the evaluation I believe the scheduler is quite poor.

The current scheduling implementation is loosely based on the Completely Fair Scheduler (CFS) found in Linux. It uses the total runtime of a process/actor combined with the priority to determine the order in which to run processes/actors. However there are many schedulers which could be a better fit for scheduling the kind of workloads found in Store*d*.

Finally overall Heph is fairly new (three years old at the time of writing) with little

[1] As mentioned in section 3.3.1 I didn't have enough time to properly evaluate the performance implication of making a single actor control the access to the datastore files.

performance engineering put into it as I've mostly focused on the design. I'm convinced there are many areas which could be improved as I've already identified a number of them, such as the scheduler data structure and the timer implementation.

Chapter 6

Related work

In this chapter I'll describe work related to this thesis.

[61] describes how "immutability changes everything", from high-level concepts such append-only applications down to the hardware level of SSDs using Copy-On-Write (COW) when writing physical blocks. It gives a good, but shallow, overview of some advantages that immutability can provide in various areas.

[62] proposes that configuration and architecture should become immutable. This would avoid "configuration drift" where the actual configuration used in production and the configuration stored (e.g. in source-control) slowly drifts apart. This can for example be caused by manual changes on production machines that are not put into the stored (source-controlled) configuration.

[63] shows how data immutability can be used in distributed data analytics, providing, among other things, better failure recovery.

[64] proposes using shared immutable byte arrays throughout network applications, enabling several valuable optimisations, and explores their strengths, weaknesses and implementation considerations.

Finally there is IPFS [65], or InterPlanetary File System, which aims to replace current web technologies such as HTTP. Instead of having central servers delivering content to all users, it uses a peer-to-peer distributed hash table (DHT) [66] with immutable objects (such as web pages or media).

6.1 Immutable Stores

In this section we'll look at some immutable data stores, some of which have already been mentioned previously.

First, Facebook's f4 [12]. f4 is a distributed immutable blob store designed to increase the storage efficiency over Haystack [67], their original blob store initially designed for storing photos.

Next there is LinkedIn's Ambry [13]. Ambry is a scalable geo-distributed object store, storing variable-sized media objects such as photos, videos and audio clips. It continually stores and serves billions of these media objects.

[68] uses immutability for archiving. It presents Deep Store, an archival store architecture that stores immutable data efficiently and reliably for multiple years (10 or even 100 years).

[69] presents Content Immutable Storage (CIS), which uses immutability to achieve trustworthy electronic record keeping. CIS is secure even against inside attacks, preventing corporate misconduct such as destroying incriminating records.

[70] argues that today's store's expose abstractions which are too low-level. They present UStore: a distributed storage with rich semantics built around a data-structure similar to that of Git [71]. It deliver three key properties: *immutability*, sharing and security.

There is also immudb [72], an immutable ledger database with built-in cryptographic proof and verification. immudb allows you to keep a record of changes in the data aiming to making it tamper-proof and auditable. It provides a key-value interface where multiple versions of the key exist as old values are not removed.

The final two stores I could find were BeanFS [73] and Ringo [14]. BeanFS is a distributed file system for a large number of immutable files. Ringo is a store which claims to have a similar design to Amazon's Dynamo [15], but lacks more documentation beyond that. More (actively developed) immutable stores I could not find at the time of writing.

6.2 Mutable Stores

As mentioned in the introduction mutable data stores have been optimised for many decades based on the structure of the data. This lead to many different kind of stores, such as SQL databases and key-value stores. In table 6.1 I've included an overview of some of these mutable data store, grouped by the model that they expose.

Note that there are many more stores available than mentioned below. For example there are many experimental/research stores which focus on special hardware capabilities such as NVM Express (NVMe) [74] or FPGA [75]. Here I've limited the mentioned to stores to those that can run on commodity hardware (which doesn't yet include NVMe or FPGAs).

Store	**On-disk/ In-memory**	**Distributed?**	**Consistency**
		SQL	
CockroachDB [76]	On-disk	Yes	Serialisable, Synchronous replication
MariaDB [77]	On-disk	Multi-master replication	Atomic changes, asynchronous replication
MySQL [78]	On-disk	Multi-master replication	Atomic changes, asynchronous replication
PostgreSQL [16]	On-disk	Multi-master replication	Configurable, can use synchronous replication
		Key-value	
Memcached [79]	In-memory	Yes	Keys are split over available machines.
Redis [5]	In-memory	Yes, via Redis Cluster	Weakly consistent with asynchronous replication
Scalaris [80]	In-memory	Yes	Strong consistency
etcd [45]	On-disk	Yes	Strong consistency, synchronous replication
		Wide-column	
Cassandra [3]	Yes	Yes	Asynchronous replication
ClickHouse [81]	Yes	Yes	Asynchronous replication
ScyllaDB [82]	On-disk	Yes	Eventual consistency
		Document	
Apache CouchDB [83]	Yes	Yes	Eventual consistency
Elasticsearch [84]	Yes	Yes	Configurable
MongoDB [85]	Yes	Yes	Causal Consistency
RethinkDB [86]	Yes	Yes	Configurable
		Multi-model	
Apache Ignite [87]	Configurable	Yes	Strongly consistent
FoundationDB [88]	Yes	Yes	Configurable
Riak [89]	Yes	Yes	Configurable

Store	**On-disk/ In-memory**	**Distributed?**	**Consistency**
Libraries			
LevelDB [90] (key-value)	Yes	No	Programmable
RocksDB [91] (key-value)	Yes	No	Programmable
SQLite [92] (SQL)	Yes	No	Strongly consistent

Table 6.1: Comparison of mutable data stores.

In short, there are many options available when it comes to mutable data stores, differing in exposed model, consistency and more factors.

Chapter 7

Conclusion

I've design and implemented a distributed immutable blob store prototype, named Store$d$, to evaluation the research questions of this thesis. The prototype is built on the Heph [21] actor framework (which I've also build) in the Rust [20] programming language.

To answer the main research question: *does taking advantage of the immutable property of data lead to a better distributed (immutable) data store?* I divided it into three sub-questions and answered them by evaluating Store$d$.

RQ1: *does taking advantage of the immutable property of data lead to a simpler design of a distributed (immutable) data store?* I couldn't find any standardised methods or evaluation tools to answer the question properly. Thus although I'm not able to proof that the using the immutable property of data leads to a simple design, I am convinced of it based on the my experience working on Store$d$.

For the next two research questions RQ2: *does a distributed store designed for immutable data perform better than stores designed for mutable data?* and RQ3: *does a distributed store designed for immutable data scale better than stores designed for mutable data?* the answer is the same, based on the evaluation of Store$d$ it does not outperform Redis or etcd in the YCSB benchmark. However I believe that the results are promising and that Store$d$ (or another immutable data store) could outperform both mutable stores by some margin, if some more performance engineering is put into it (some of which is discussed in section 5.4).

References

[1] David Reinsel-John Gantz-John Rydning. **The digitization of the world from edge to core**. *Framingham: International Data Corporation*, 2018. i, 1

[2] **NoSQL Database Management Systems**. `https://nosql-database.org`, 2020. 1

[3] **Apache Cassandra**. `https://cassandra.apache.org`, 2020. 1, 60

[4] **Neo4j Graph Platform**. `https://neo4j.com`, 2020. 1

[5] **Redis**. `https://redis.io`, 2020. 1, 3, 34, 60

[6] **The Rust community's crate registry**. `https://crates.io`, 2020. 1, 3

[7] **npm Node.js package manager**. `https://www.npmjs.com`, 2020. 1, 3

[8] Jim Gray and Leslie Lamport. **Consensus on transaction commit**. *ACM Trans. Database Syst.*, **31**(1):133–160, 2006. 2, 8

[9] Diego Ongaro and John K. Ousterhout. **In Search of an Understandable Consensus Algorithm**. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014. 2, 8, 37

[10] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. **Spanner: Becoming a SQL System**. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 331–343. ACM, 2017. 2, 7

[11] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. **Yesquel: scalable sql storage for web applications**. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 245–262. ACM, 2015. 2

[12] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. **f4: Facebook's Warm BLOB Storage System**. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 383–398. USENIX Association, 2014. 3, 4, 7, 58

[13] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. **Ambry: LinkedIn's Scalable Geo-Distributed Object Store**. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 253–265. ACM, 2016. 3, 4, 7, 34, 59

[14] **Ringo**. `https://github.com/tuulos/ringo`, 2020. 3, 59

[15] 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 Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007. 3, 59

[16] **PostgreSQL: The world's most advanced open source database**. `https://www.postgresql.org`, 2020. 3, 6, 60

[17] **Docker Hub**. `https://hub.docker.com`, 2020. 3

[18] Nat Friedman. **npm is joining GitHub**. `https://github.blog/2020-03-16-npm-is-joining-github/?utm_campaign=1584377606&utm_medium=social&utm_source=twitter&utm_content=1584377606`, 2020. 3

[19] Thomas de Zeeuw. **Stored: a distributed immutable blob store — GitHub**. `https://github.com/Thomasdezeeuw/stored`, 2020. 4

[20] The Rust Project Developers. **Rust Programming Language**. https://www.rust-lang.org, 2020. 4, 12, 62

[21] Thomas de Zeeuw. **Heph — GitHub**. https://github.com/Thomasdezeeuw/heph, 2020. 4, 12, 62, 73

[22] National Institute of Standards and Technology. **FIPS PUB 180-4**. *Federal Information Processing Standards. FIPS*, 2015. 6, 38

[23] **PgBouncer - lightweight connection pooler for PostgreSQL**. https://www.pgbouncer.org, 2020. 6

[24] **nginx**. https://nginx.org, 2020. 7

[25] **HAProxy — The Reliable, High Performance TCP/HTTP Load Balancer**. https://www.haproxy.org, 2020. 7

[26] Jim Gray. **Notes on Data Base Operating Systems**. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors, *Operating Systems, An Advanced Course*, **60** of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978. 8, 20, 38

[27] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. **Causal Memory: Definitions, Implementation, and Programming**. *Distributed Comput.*, **9**(1):37–49, 1995. 8, 14

[28] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach.* The MIT Press, 2 edition, 2018. 8, 14

[29] **Fsync Errors - PostgreSQL wiki**. https://wiki.postgresql.org/wiki/Fsync_Errors, 2020. 11

[30] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. **A Universal Modular ACTOR Formalism for Artificial Intelligence**. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973. 12, 72

[31] Wikipedia contributors. **Actor model — Wikipedia**. https://en.wikipedia.org/wiki/Actor_model, 2020. 12, 72

[32] **aio(7) — Linux manual page**. https://man7.org/linux/man-pages/man7/aio.7.html, 2020. 12

[33] Jens Axboe. **Efficient IO with io_uring**. https://kernel.dk/io_uring.pdf, 2019. 12

[34] Ka-Hing Cheung. **How we scaled nginx and saved the world 54 years every day**. https://blog.cloudflare.com/how-we-scaled-nginx-and-saved-the-world-54-years-every-day, 2018. 12

[35] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. **ffwd: delegation is (much) faster than you think**. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 342–358. ACM, 2017. 13

[36] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. **Noria: dynamic, partially-stateful data-flow for high-performance web applications**. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 213–231. USENIX Association, 2018. 13, 56

[37] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. **Anna: A KVS for Any Scale**. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 401–412. IEEE Computer Society, 2018. 13

[38] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **WiscKey: Separating Keys from Values in SSD-conscious Storage**. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 133–148. USENIX Association, 2016. 14

[39] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking cloud serving systems with YCSB**. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. 33

[40] **Yahoo! Cloud Serving Benchmark on GitHub**. https://github.com/brianfrankcooper/YCSB, 2020. 33

[41] Kim Shanley. **TPC Releases New Benchmark: TPC-C**. *SIGMETRICS Perform. Evaluation Rev.*, **20**(2):8–9, 1992. 33

[42] **TPC-C**. `http://www.tpc.org/tpcc`, 2020. 33

[43] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John W. Romein, Frank J. Seinstra, Cees Snoek, and Harry A. G. Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *IEEE Computer*, **49**(5):54–63, 2016. 33

[44] **DAS-5: Distributed ASCI Supercomputer 5**. `https://www.cs.vu.nl/das5/home.shtml`, 2020. 33

[45] **etcd**. `https://etcd.io`, 2020. 34, 60

[46] **Flexible I/O Tester**. `https://github.com/axboe/fio`, 2020. 34

[47] **Tmpfs**. `https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt`, 2020. 34

[48] **RAM disk block device**. `https://www.kernel.org/doc/Documentation/blockdev/ramdisk.txt`, 2020. 34

[49] **Redis Cluster Tutorial**. `https://redis.io/topics/cluster-tutorial`, 2020. 36

[50] **Redis Cluster Specification**. `https://redis.io/topics/cluster-spec`, 2020. 36

[51] **Redis Replication**. `https://redis.io/topics/replication`, 2020. 36

[52] **Jedis**. `https://github.com/redis/jedis`, 2020. 36

[53] **connect(2) — Linux manual page**. `https://man7.org/linux/man-pages/man2/connect.2.html`, 2020. 36

[54] **Linux kernel overcommit policy**. `https://www.kernel.org/doc/Documentation/vm/overcommit-accounting`, 2020. 36

[55] **epoll - I/O event notification facility**. `https://man7.org/linux/man-pages/man7/epoll.7.html`, 2020. 53

[56] Anuj Kalia, Michael Kaminsky, and David G. Andersen. **FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs**. In Kimberly Keeton and Timothy Roscoe, editors, *12th*

USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, pages 185–201. USENIX Association, 2016. 55

[57] Anuj Kalia, Michael Kaminsky, and David G. Andersen. **Using RDMA efficiently for key-value services**. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014. 55

[58] Mark J. Eisner and Dennis G. Severance. **Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases**. *J. ACM*, **23**(4):619–635, 1976. 55

[59] Pedro Ramalhete and Andreia Correia. **Left-Right: A Concurrency Control Technique with Wait-Free Population Oblivious Reads**. `http://sourceforge.net/projects/ccfreaks/files/papers/LeftRight/leftright-extended.pdf`, 2013. `https://concurrencyfreaks.blogspot.com/2013/12/left-right-concurrency-control.html`. 56

[60] Jon Gjengset et al. **evmap — A lock-free, eventually consistent, concurrent multi-value map**. `https://github.com/jonhoo/rust-evmap`, 2020. 56

[61] Pat Helland. **Immutability Changes Everything**. *ACM Queue*, **13**(9):40, 2015. 58

[62] Anders Mikkelsen, Tor-Morten Grønli, and Rick Kazman. **Immutable Infrastructure Calls for Immutable Architecture**. In Tung Bui, editor, *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8-11, 2019*, pages 1–9. ScholarSpace, 2019. 58

[63] Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu. **Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics**. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 191–206. USENIX Association, 2019. 58

[64] Dmitri Nikulin. **Shared Immutable Byte Arrays For Distributed Applications**. 58

[65] Protocal Labs. **InterPlanetary File System (IPFS)**. `https://ipfs.io`, 2020. 58

[66] Wikipedia contributors. **Distributed hash table — Wikipedia**. `https://en.wikipedia.org/wiki/Distributed_hash_table`, 2020. 58

[67] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. **Finding a Needle in Haystack: Facebook's Photo Storage**. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 47–60. USENIX Association, 2010. 58

[68] Lawrence You, Kristal T. Pollack, and Darrell D. E. Long. **Deep Store: an Archival Storage System Architecture**. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 804–815. IEEE Computer Society, 2005. 59

[69] Lan Huang and Fengzhou Zheng. **CIS: Content Immutable Storage for Trustworthy Electronic Record Keeping**, 2004. 59

[70] Anh Dinh, Ji Wang, Sheng Wang, Gang Chen, Wei-Ngan Chin, Qian Lin, Beng Chin Ooi, Pingcheng Ruan, Kian-Lee Tan, Zhongle Xie, Hao Zhang, and Meihui Zhang. **UStore: A Distributed Storage With Rich Semantics**. *CoRR*, **abs/1702.02799**, 2017. 59

[71] **Git**. `https://git-scm.com`, 2020. 59

[72] CodeNotary. **immudb - CodeNotary**. `https://codenotary.com/technologies/immudb`, 2020. 59

[73] Wook Jung Daewoo Lee Eunji Pak, Youngjae Lee Sang-Hoon Kim, Jin-Soo Kim, Taewoong Kim, and Sungwon Jun. **BeanFS: A Distributed File System for a Large Number of Immutable Files**, 2008. 59

[74] Inc NVM Express. **NVM Express — scalable, efficient, and industry standard**. `https://nvmexpress.org`, 2020. 59

[75] Wikipedia contributors. **Field-programmable gate array — Wikipedia**. `https://en.wikipedia.org/wiki/FPGA`, 2020. 59

[76] Cockroach Labs. **CockroachDB**. `https://www.cockroachlabs.com/product`, 2020. 60

[77] MariaDB contributors. **MariaDB Enterprise Open Source Database**. `https://mariadb.com`, 2020. 60

[78] MySQL contributors. **MySQL**. https://www.mysql.com, 2020. 60

[79] **memcached - a distributed memory object caching system**. https://memcached.org, 2020. 60

[80] **Scalaris**. http://scalaris.zib.de, 2020. 60

[81] **Clickhouse**. https://clickhouse.tech, 2020. 60

[82] **ScyllaDB**. https://www.scylladb.com, 2020. 60

[83] **CouchDB**. https://couchdb.apache.org, 2020. 60

[84] **Elasticsearch**. https://www.elastic.co/products/elasticsearch, 2020. 60

[85] **MongoDB**. https://www.mongodb.com, 2020. 60

[86] **RethinkDB**. https://rethinkdb.com, 2020. 60

[87] **Apache Ignite**. https://ignite.apache.org, 2020. 60

[88] **FoundationDB**. https://www.foundationdb.org, 2020. 60

[89] **Riak KV**. https://docs.riak.com/riak/kv/latest/index.html, 2020. 60

[90] **LevelDB**. https://github.com/google/leveldb, 2020. 61

[91] **RocksDB**. https://rocksdb.org, 2020. 61

[92] **SQLite**. https://sqlite.org/index.html, 2020. 61

[93] Wikipedia contributors. **Binary large object — Wikipedia**. https://en.wikipedia.org/wiki/Binary_large_object, 2020. 72

[94] Wikipedia contributors. **Consensus (computer science) — Wikipedia**. https://en.wikipedia.org/wiki/Consensus_(computer_science), 2020. 72

[95] Wikipedia contributors. **Coroutine — Wikipedia**. https://en.wikipedia.org/wiki/Coroutine, 2020. 72

[96] Wikipedia contributors. **Green threads — Wikipedia**. https://en.wikipedia.org/wiki/Green_threads, 2020. 73

[97] Wikipedia contributors. **Hash table — Wikipedia**. https://en.wikipedia.org/wiki/Hash_table, 2020. 73

[98] Wikipedia contributors. **Hypertext Transfer Protocol — Wikipedia**. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, 2020. 73

[99] Wikipedia contributors. **Node (networking) — Wikipedia**. https://en.wikipedia.org/wiki/Node_(networking), 2020. 73

[100] Wikipedia contributors. **NoSQL — Wikipedia**. https://en.wikipedia.org/wiki/NoSQL, 2020. 73

[101] Wikipedia contributors. **Remote procedure call — Wikipedia**. https://en.wikipedia.org/wiki/Remote_procedure_call, 2020. 73

[102] Wikipedia contributors. **SQL — Wikipedia**. https://en.wikipedia.org/wiki/SQL, 2020. 74

[103] Wikipedia contributors. **Stable storage — Wikipedia**. https://en.wikipedia.org/wiki/Stable_storage, 2020. 74

[104] Wikipedia contributors. **URL — Wikipedia**. https://en.wikipedia.org/wiki/URL, 2020. 74

Glossary

actor

An actor in the actor model, see actor model. 12, 18, 38

actor model

The actor model in computer science is a mathematical model of concurrent computation that treats actor as the universal primitive of concurrent computation. In response to a message it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through messaging (removing the need for lock-based synchronisation). [30, 31]. 12

blob

A Binary Large OBject (BLOB) is a collection of binary data stored as a single entity in a database management system [93]. 3, 4, 6, 14, 18–20, 23, 24, 28, 30, 31, 34, 35, 58

consensus

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation [94]. 2, 12, 37, 38

coroutine

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed [95]. 13

green thread

In computer programming, green threads are threads that are scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system (OS) [96]. 13

HashMap

In computing, a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values [97]. 16

Heph

Heph is an actor framework for Rust based on asynchronous functions. [21]. 4, 12, 32, 44

HTTP

The Hypertext Transfer Protocol (HTTP) is an application layer protocol for distributed, collaborative, hypermedia information systems [98]. 6, 13, 18, 38

node

A communication endpoint in a network capable of creating, receiving, or transmitting information over a communications channel. [99]. 3, 12, 18–20, 73

NoSQL

A NoSQL (originally referring to "non-SQL" or "non-relational") database provides a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases [100]. 1

peer

A node A that is connected to node B, considers node B its peer. v, 8, 12, 14, 19, 20, 24, 31, 35–37

RPC

In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction [101]. 13, 14

SQL

Structured Query Language (SQL) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS) [102]. i, 1

stable storage

Stable storage is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures. [103]. 14, 34, 38

URL

A Uniform Resource Locator (URL), colloquially termed a web address, is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it [104]. 26, 36