Log-Based Distributed Key-Value Stores

João Bernardo Sena Amaro
joaobsamaro@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. A key-values store is a fundamental building block in today's applications. It allows to persistently store and retrieve data that is indexed by a primary key. Key-values stores have been widely used to support internet applications given that its interface allows for highly decentralized implementations, that can scale to millions of users. Early key-value stores offered weak consistency guarantees and no support for multi-key operations, such as transactions. However, there is a growing interest in using key-value stores that offer a richer set of properties. Among the different alternatives to build such key-value stores, the construction of a key-value stored over a shared log as been recently explored and a viable alternative. The focus of this report is on log based key-value storage systems, more specifically on the problem of scaling them while providing strong semantics to applications. We identify two main design to build such systems, namely sequencer based architectures and sequencer free architectures, and we propose to explore the possibility of merging the benefits of both, to increase the system scalability while preserving strong semantics.

1 Introduction

Most computer applications require the access to some form of persistent storage service. Many persistent stores exist, from simple file systems to complex databases. Key-value stores have emerged as a sweet-spot between functionality and ease of implementation, and have become the technology of choice for application that need to scale to millions of users. Typically, key-value stores do not offer transactional support for operations that require reading and updating multiple keys. However, such functionality is often needed by applications, and this raised the interest of extending key-value stores with transactional support.

Among the different designs that can be used to extend key-value stores with transactional support, log-based key-values stores emerged recently as a viable alternative. Such designs have been made possible due to the emergence of several highly-efficient shared log systems, such as CORFU [1]. In this work we address log based key-value storage systems, more specifically we focus on the problem of scaling them while providing strong semantics to applications. We identify two main design to build such systems, namely synchronous sequencer based architectures and asynchronous sequencer free architectures. Synchronous

systems tend to provide stronger semantics but are harder to scale while asynchronous systems provide weaker semantics but should offer better scalability. We propose to explore the possibility of merging the benefits of both, to increase the system scalability while preserving strong semantics.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present relevant work that has been done in the area of log based storage systems. Section 4 describes the proposed solution and Section 5 describes how we plan to evaluate it. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of building scalable log based key-value storage systems that provide strong semantics to applications. More precisely, we aim at:

> *Goals:* Designing and implementing a novel log based key-values storage system that provides causal consistency and support for ACID transactions.

As will be described further ahead in the report our system will try to merge the benefits of two distinct architectures, the strong semantics of synchronous systems with the scalability benefits of asynchronous systems. We will focus on the performance and scalability of the proposed system and will assess its suitability for the existing applications by performing an experimental evaluation.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the system; ii) an implementation of a prototype of the system, and iii) an extensive experimental evaluation of the performance and scalability of the system using real world workloads.

3 Related Work

This section surveys relevant work that has been produced in the area of log based storage systems and is organised as follows. Section 3.1 introduces the concept of a shared log. Section 3.2 starts by addressing log based storage systems and their properties. Then Section 3.2.1 and Section 3.2.2 present examples of these systems. Finally, Section 3.3 discusses and compares the systems described in the previous sections.

3.1 Shared Log

A log is an append-only, totally-ordered, sequence of immutable records ordered by time. A *shared* log can be accessed by multiple clients, that can read and write to it concurrently. Writing to the log is done by appending a record to the end of the log, and reading from it can be performed in two ways, by scanning a range of log records, for example all the records from left to right in Figure 1, or by reading specific log records by their position in the log, like reading log record number 3 directly.

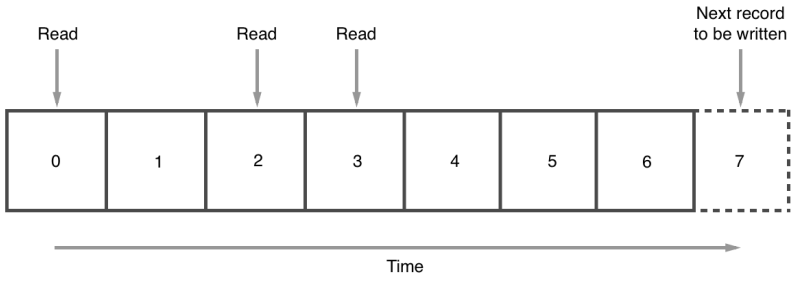

Fig. 1: Log

The two fundamental properties of a log are the persistence of data records, and the fact that these data records are totally ordered. Many high-level tasks in distributed systems require operations to be totally ordered, for instance, transactional systems require concurrent transactions to be serialized, and state-machine replication, a fundamental technique to achieve fault-tolerance in distributed systems, requires commands to be totally-ordered. Therefore, a shared log implements some of the key ingredients for several distributed applications and has the potential to simplify the implementation of transactional key-value stores.

The implementation of a shared log requires a solution to two fundamental problems:

1. Contention Problem: created at the tail of the log by clients that want to append a record at the same time. Clients all want to append a record to the tail, but only one will win.
2. Playback Bottleneck Problem: due to the append-only nature of the shared log, its size keeps growing as new records are appended. If no mechanism is used to mitigate this problem, clients that need to playback the log from the beginning will face a *playback bottleneck* problem, because they will not be

able to keep up with the rate of new appends and thus consume the whole log.

The first problem can be summarized as a position assignment problem, where logs have to assign positions to records. In order to solve this problem, the log design that exist today have diverged into two different solutions:

- Pre-assignment: is a technique used by some log designs, that assigns log positions to records before they have been sent to the persistence layer. This means that clients know the log position where their record is going to be persisted before they sent it to the persistence layer. This technique usually requires a sequencer that is responsible for log positions to clients before they append a record.
- Post-assignment: is a technique used by some log designs, that assigns log positions to records after they have been sent to the persistence layer. This means that clients do not know the log position where their record is going to be persisted before they sent it to the persistence layer. This technique usually requires a serialization procedure to be run regularly to serialize all the appended records.

When it comes to the second problem, there are very different solutions available, but as in the contention problem there are two distinct trends to solve it. The first trend is to ignore the problem and rely on a regular garbage collection of the log to keep it in a consumable size. And the second trend is to index log records, allowing applications to consume only a subset of the records in the log.

It is worth noting that, although the total order of log entries is a characteristic of most long implementations, a few systems relax this property, implementing only a causally ordered sequence of log entries, as a means to reduce the contention that may exist when multiple clients try to append updates to the shared log concurrently. When appropriate, we also discuss these alternative designs.

For simplicity reasons, from this point on in the report, we will refer to a *shared log* as a *log*.

3.2 Log Based Storage Systems

Log based storage systems are as the name implies storage systems built over a log. These systems exploit the consistency guarantees and scalability of logs to build equally consistent and scalable storage systems.

These systems can be classified according to the functional requirements they provide to applications, using the following criteria:

- Synchrony: distinguishes systems by how they update the log. Synchronous systems make client operations visible in the log *right* when they run, while asynchronous systems update the log with the client operations *after* they run.

– Log Order: as stated in Section 3.1 a log is a totally-ordered sequence of records, but achieving a total order is expensive and not always needed. Some systems opt to relax the log order to a causal order or a total order per shard to be able to scale better.
– Support for Transactions: this criterion distinguishes systems that have support for ACID read-write transactions from systems that do not.
– Log Indexing: this criterion distinguishes systems that index log records from those that do not. And for those that implement some kind of indexing this criterion specifies how the indexing is done.

When it comes to the non functional requirements, log based storage systems can be classified according the following criteria:

– Replication: is the ability of a system to create more than one independent copy (replica) of the data it stores. A storage system can either be replicated or not.
– Sharding: is the ability of a system to store data in more than one machine per replica. A storage system can either be sharded or not.

In Section 3.2.1 and Section 3.2.2 we will use the *synchrony* criterion as the main one to differentiate the systems that are going to be presented, because it is tightly related with their implementation and design.

3.2.1 Synchronous Systems

CORFU [1] is a totally ordered log abstraction built over a cluster of flash storage units.

CORFU's architecture is composed by a sequencer and an array of flash units. Flash units are treated as passive storage devices that are accessed over the network and the sequencer is used as a *pre-assignment* technique, that is responsible ordering log records. In CORFU the ordering of records is decoupled from their persistence, allowing parallelism when appending to the flash units.

The functionality of CORFU is all made available to clients in the form of a client library. It works by implementing three fundamental functions:

– Mapping function: responsible for mapping logical log positions to flash pages in the cluster of flash units.
– Tail finding mechanism: for finding the next available logical position in the log.
– Replication protocol: to persist log records in several flash pages in the flash unit cluster.

The mapping function works by having each client maintain a local read-only copy of a structure called a *projection*. The projection structure splits the log into disjoint ranges, and each range is mapped to a list of extents within the address space of individual flash units. In essence, a projection is a mapping of logical addresses space to flash pages. When a flash unit fails or the log grows past the

maximum available position in the projection a reconfiguration mechanism takes place to install a new projection in all the clients.

To treat the address space as an appendable log, clients must be able to find the tail of the log and write to it. CORFU implements two tail finding mechanisms:

- Contention mechanism: relies on the flash unit *write-once* semantics that CORFU requires. Clients that wish to append records to the log try to concurrently write to the first position. One client wins, while the rest fails. The client that wins successfully appends to the log while the others have to try again in in the next position.
- Sequencer: is used in CORFU as an optimization over the contention mechanism. The sequencer is responsible for assigning empty log positions to clients. A client that wants to append a record to the log, first contacts the sequencer to get a log position, once he receives the position from the sequencer, he uses his local projection to find the flash unit responsible for storing that log position and persists the record there without contention from other clients.

When writing to a log position, CORFU clients use a chain replication protocol. It works in two steps: (1) clients use their local projections to map a single log position to a set of flash pages, and (2) write to this set of flash pages in a deterministic order waiting for each flash unit to respond before moving to the next one. The write is successfully completed when the last flash unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of flash pages, one of them will arrive second at the first unit of the chain and receive an overwritten error.

CORFU's design ensures that the log throughput is not a function of the I/O bandwidth of any single flash unit, instead clients can append records as fast as the sequencer can assign positions.

Tango + CORFU [2] is a log based object store intended to be used as a building block of highly available metadata services. Objects stored in Tango are replicated, in-memory data structures that have their state durably stored in a log. Tango uses CORFU as a log, taking advantage of all the properties that it provides.

The state of a Tango object exists in two forms, a *history*, which is an ordered sequence of updates stored durably in the CORFU log, and several *views*, that are full or partial copies of an object stored in the memory of the clients. Because the state of Tango objects is stored as an ordered sequence of updates, application developers can roll back to any point in the history of an object simply by creating a new instance and syncing it with the appropriate updated of the log.

The implementation of a Tango object requires three three main components. The first is a view of the object, which is an in memory representation of the object's state. The second is an *apply()* function, responsible for updating the object's view when there are new updates in the log. And the third is the object's

public interface with mutator and accessor methods. Mutator methods do not change the object's view directly, instead they append the mutation to the log. When accessor methods are called, they first get the latest mutations from the log, update the object's view using the *apply()* function and only then apply the accessing logic.

The log based design of Tango allows strongly consistent operations across objects to be achieved trivially, by just using reads and appends on the log. But the authors of Tango went a step further and implemented an optimistic concurrency control system over the log, that allows applications to run transactions across objects. It works by appending a speculative commit record to the log that ensures atomicity of the transaction. It marks a point in the total ordering of updates at which the changes of the transaction can be made visible. Each commit record includes a *read set*, a list of object read during the transaction, that is is used to ensure transaction isolation. A transaction succeeds if the objects in the read set have not been changed since they were read.

Having all the object's state in the same log is important to offer strongly consistent operations and transactions, however it might introduce the *playback bottleneck* problem. If client just want to host views of certain objects, consuming updates from all the objects can be wasteful. To overcome this problem Tango implements objects streams over a single log, allowing clients to selectively consume the log updates corresponding to certain objects. This means that clients will host a layered partition of the log with all the same strongly consistent and transactional semantics of the full log.

To allow clients to consume the log via a streaming interface the authors proposed a modification to the underlying CORFU design, a stream header that is present in every log record. This stream header includes the stream ID as well as backpointers to the last k records of that stream, allowing clients to construct a linked list of records from the same stream. Reading and appending to streams work as before, the only difference is in the appends that require the client to build the stream header before appending a record.

vCorfu [3] is a strongly consistent cloud-scale object store built over a log just like Tango + CORFU, that uses a *stream materialization* technique to overcome the playback bottleneck problem that logs have.

Stream materialization is a technique that enables the design of systems that store large quantities of state in the form of a log without introducing the playback bottleneck problem to clients. Stream materialization is a step ahead Tango's streams, which are implemented as simple tags on the log, materialized streams are a first class abstraction which support random and bulk reads. vCorfu uses a materialized stream per object it stores.

Log appends to a materialized stream are stored in two replicas, the *log replicas* and the *stream replicas*. These two replicas store the same data, but unlike Tango + CORFU the data is indexed in different ways. Log replicas index the log records according to their position on a global log, while stream replicas store log records indexed by their position in the stream log. This replication

scheme, inspired in the idea of Replex [4] of using different replicas to store the same data indexed in different ways, allows clients to directly read the latest version of an object by simply contacting the corresponding stream replica.

Materialized streams can be seen as independent logs. Appending to them is very similar to appends in Tango, the difference introduced is in the client-sequencer communication. Clients that contact the sequencer receive two log positions instead of one. One corresponding to the global log and one corresponding to the stream log. Using the two log positions clients persist the record in both replicas using a chain replication protocol. Because appends to these *stream logs* have a corresponding append in the global log, there is a total order over all appends from all the streams, enabling vCorfu to support atomic appends across streams (objects).

As Tango, vCorfu went a step further the atomic updates and implemented a transactional system that enables application to run transactions across objects. Transactions are implemented by combining the atomic update capabilities with the optimistic concurrency control techniques used in Tango. The sequencer is exploited as a lightweight transaction manager, but unlike Tango it only issues log positions to commit a transaction if the transaction has no conflicts, an important optimization that frees clients from the transaction validation logic every time they playback the log.

The vCorfu system can be seen as a new version of Tango, a version with all the same functionality but with a better implementation and performance. On the other hand this also means that vCorfu suffers from the same problem that Tango does, the sequencer I/O bottleneck. Chariots and Kafka are examples of systems that avoid this problem by not using a sequencer, an overview of how they do it is ahead in the report.

Megastore [5] is a log based storage system that blends the scalability of NoSQL data stores with the convenience of a traditional RDBMS, providing both high availability and strong consistency guarantees.

The interface and data model provided by Megastore is very similar to the one provided by a relational database, applications can create schemas, tables and indexes, that can later be queried using a SQL like language. An addition that Megastore introduced over the conventional relational databases is the ability to partitioned the data in what the authors call *entity groups*. Entity groups are an a priori grouping of related data for fast operations that can be seen as small databases with ACID semantics.

Each entity group has its own write-ahead log, where all the updates that happen inside it are recorded. The log is independently and synchronously replicated over a wide area, and is stored in Bigtable [6], leveraging its scalability and fault-tolerance capabilities. Updates inside entity groups are done as single phase ACID transactions, while updates across entity groups require expensive two phase-commit protocols or Megastore's asynchronous messaging system.

When it comes to transactions inside an entity group, their life cycle is the following. First the client obtains the timestamp of the latest committed trans-

action from the log, then reads from a consistent snapshot using the timestamp gathered in the first step and writes into a log record. Finally, in order to commit the transaction the client appends the log record with all the writes to the end of the log using the Paxos algorithm. After the record is appended to the log the updates are applied to the data in Bigtable.

To allow clients to read from a consistent snapshot, Megastore relies on Bigtable's capability of storing multiple values for the same row/column pair. This makes the implementation of multi version concurrency control trivial: updates within a transaction are written with the timestamp of the transaction while concurrent reads that use lower timestamps never see partial updates. In other words reads are isolated from the writes.

Appending to the log is done by running an independent instance of the Paxos algorithm for each log position, so when a client wants to append a record to the log, he has to propose the record he wants to append as the one to be placed at the tail of the log, and has to block until he gets an answer from the majority of the replicas. If another client takes the position first, he has to abort and start again in the next log position. Because Paxos is responsible for assigning log positions to clients it can be seen as the log sequencer.

Transactions across entity groups are also possible, using Megastore's asynchronous messaging system. Entity groups can communicate between them in an RPC style, each entity group has an inbox to where other entity groups can send messages. A client that wants to update several entity groups, runs a transaction to atomically send updates to several entity group's inboxes. Each entity group will then process the updates they receive as an isolated transaction.

3.2.2 Asynchronous Systems

Chariots [7] is a highly available, geo-replicated, causally-ordered log that overcomes I/O bottlenecks of existing sequencer based log designs.

The Chariots paper includes two main contributions:

- FLStore: sequencer free log store.
- Chariots: multi stage log replication pipeline.

FLStore or Fractal Log Store is a log store that uses a post-assignment technique to assign log positions to clients. It consists of two groups of servers, the *log maintainers* and the *indexers*. Log maintainers are responsible for disjoint ranges of the log, for each range they store log records, they serve read requests and they assign log positions. Indexers are responsible for indexing log record's tags.

The post assignment technique works based on the fact that log maintainers are responsible for distinct ranges of the log, which means that a log position can only be assigned, written and stored by a single log maintainer. Appending to the log works by sending a record to a log maintainer at random, the log maintainer will assign it the next available position in the range that he is responsible for and

the record is persisted in that position. This capability of assigning log positions without coordinating with others is a big improvement over CORFU, because the append throughput is now a function of the aggregate I/O bandwidth of *all* log maintainers and not a function of the I/O bandwidth of a *single* machine.

On the other hand, allowing clients to contact any log maintainer at random introduces a couple of new problems. The first is the existence of holes at the tail of the log, which may delay the visibility of certain records. This happens because reading a certain log position is only allowed when all the previous log positions are filled up. Consider the case of an append to position 15 of the range 10-19, if the range 0-9 is empty or is not yet full, the record appended to position 15 will not be available for reading. The second problem introduced by this design is the lack of explicit order between records appended by the same client. For example, when a client appends two records sequentially, he can not know the order between them in the log, even though they were added sequentially. When it comes to solving these problems, no solutions exist for the first problem, and for the second one two solutions are described in the paper but they all have some drawbacks that might impact the client or the system performance.

Chariots, the second main contribution of the paper, is a log replication pipeline designed to replicate a log to new geographic locations, but its ideas can also be applied to local replication. It runs over FLStore and is responsible for processing all append requests. FLStore is used as the persistence layer in the pipeline. Chariots' design favours availability of the log by relaxing the log consistency, providing applications with a causally ordered log.

The Chariots pipeline works as follows. When a client wants to append a record to the log, sends a request with the record to append to Chariots. The record enters the pipeline, is tagged with its causal dependencies and is moved into a queue where it is assigned a position in the log. After having a position assigned, the record is sent to FLStore, more precisely to the log maintainer responsible for the position that it was assigned. Finally after being persisted in FLStore, the senders from Chariots catch it and send it to other Chariots instances for them to incorporate it.

Appends received from remote data centres, remote appends, go through the pipeline as a local append would, the main difference is in the queue stage, where their causal dependencies are verified. If they are met, the record can be inserted in the local log, if they are not met the record is kept in the queue until they are.

Kafka [8] is a log based messaging system that was developed for collecting and delivering high volumes of data with low latency. It combines the benefits of log aggregators and messaging systems to build a system that allows applications to consume data in real time.

As every messaging system, Kafka's interface and data model is around messages. The smallest data storage unit in Kafka is called a *message*, messages are stored in *topics*, which are aggregations of messages of a certain type. Topics are used later to retrieve messages according to their type. The API provided

to applications is very similar to a publish subscribe system, clients can publish messages to topics and can later subscribe to them to read the messages.

When it comes to the storage of messages, they are stored in *brokers*. Inside each broker they are stored per topic, as a log on disk. In order to sustain high volumes of messages, topics can be partitioned. At the storage level this corresponds to the creation of independent logs for the same topic. Having independent logs means that no order exists between them, a limiting factor for some applications. All this behaviour makes brokers very similar to log maintainers in Chariots, with the small difference that brokers are responsible for completely independent logs and not ranges of a single one.

Publishing a message in Kafka is very simple, a client randomly chooses one of the partitions of the topic where he wishes to publish a message, finds which broker is holding the partition and sends the message to that broker. The broker receives the message and simply appends it to the corresponding log. Just like in Chariots brokers can assign log positions from the logs they are responsible for without entering in any coordination with other brokers, which makes each broker a sequencer of the logs they hold.

In order to read messages from a given topic, clients have to join what is called in Kafka a *consumer group*. A consumer group is a group of one or more consumers that jointly consume a set of subscribed topics. To avoid locking mechanisms and state management Kafka makes the partition of a topic the smallest unit of parallelism, i.e. at any given time all the messages of a partition are only consumed by a single consumer within a group. Messages from a partition are always consumed sequentially.

As described before, Kafka does not require any coordination for publishing messages, however, consuming messages requires some level of coordination between consumers in a consumer group. For this, Kafka does not rely in a central node, consumers coordinate among themselves in a decentralised fashion. To facilitate the coordination a consensus service called Zookeeper [9] is used for de detecting the addition and the removal of brokers and consumers, triggering a rebalance process in each consumer when necessary, maintaining the consumption relationship between partitions and consumer, and keeping track of the consumed offset of each partition.

Even though Kafka has some similarities with Chariots, its design is optimized for delivering messages, which results in a log based storage system that provides applications with a partitioned log with a total order per partition.

Eunomia [10] is an event ordering service that produces an causally ordered log of events.

In the paper, Eunomia is presented as a building block of a key value store called EunomiaKV, where it is used as an update serializer. It works in the background serializing all the updates occurring in the store. The result of the serialization is a causally ordered log of updates that is used to replicate the updates to new geographic locations. In our description of the system we will

ignore the geographic replication capabilities of EunomiaKV, instead we will focus on the log building techniques used.

The EunomiaKV architecture is composed by *partitions* and the *Eunomia* service. Partitions are responsible for storing ranges of key value pairs and serving read and update requests on the them. The Eunomia service receives all the updates that each partition handles and orders them before sending them to new geographic locations.

Read requests handled by the partitions, correspond to simple key value fetches on the underlying storage layer, returning to the client the value of they key requested and the timestamp it is tagged with. The client uses the timestamp returned partition to update its local clock, which is responsible for keeping track of his causal dependencies. Update requests are tagged with timestamps, representing the causal dependencies of both the client and the partition, are applied locally and are sent to Eunomia to be serialized.

At Eunomia, updates received from the partitions are placed in a *non-stable* updates queue. Regularly Eunomia runs a *process stable* routine that finds a set of stable updates in the non-stable updates queue, removes them and orders them in timestamp order, producing a causally ordered log of updates. We should note that the log produced by Eunomia has a significant difference when compared to the log produced by all the systems presented. It is a log composed only by ids, this means that the log does not include the content of the updates. This happens because the only thing that partitions send to Eunomia are the update id and its timestamp. A significant difference that avoids overloading the network.

Allowing partitions to serve read and update requests without entering in synchronous coordination with any other component in the system, is a fundamental characteristic of the Eunomia design. It does not limit the system throughput like sequencer based designs do, because the load is distributed across all the partitions, and the latency of operations is decreased, because partitions can reply to client without having to wait from a response from other component. On the other hand, this design introduces one main trade-off when compared to sequencer based designs, an increase of the visibility latencies of the log, i.e. the log takes longer to be available for reading.

3.3 Systems Comparison

Table 1 summarizes the log based storage systems presented.

Section 3.2.1 and Section 3.2.2 presented two distinct types of log based storage systems, synchronous and asynchronous. As shown in Table 1, all the systems tend to agree on the importance of data replication and sharding, the main difference between them is in the *Sequencer* column, synchronous systems have a sequencer, while asynchronous systems do not.

Synchronous systems (CORFU, Tango, vCorfu and Megastore) have the advantage of being easier to implement and reason about. The centralized sequencer makes it easy to provide a total order across the log, synchrony when appending records and operations with strong semantics, like atomic updates and transactions. However, because the sequencer is on the critical path of clients it has

System	Synchrony	Sequencer	Log Order	Transactions	Log Indexing	Replication	Sharding
CORFU	Synchronous	Yes	Total	No	No	Local	Yes
Tango	Synchronous	Yes	Total	Yes	Backpointers	Local	Yes
vCorfu	Synchronous	Yes	Total	Yes	Materialized	Local	Yes
Megastore	Synchronous	Yes	Total per partition	Yes	No	Both	Yes
Chariots	Asynchronous	No	Causal	No	Tags	Geo	Yes
Kafka	Asynchronous	No	Total per partition	No	No	No	Yes
Eunomia	Asynchronous	No	Causal	No	No	Geo	Yes

Table 1: Systems comparison table

some trade-offs: it can be seen as a single point of failure, it may limit the system throughput if its I/O bandwidth becomes saturated and may limit the ability to scale the system to different geographic locations.

On the other hand, asynchronous systems (Chariots, Kafka and Eunomia) offer different trade-offs: they do not suffer from any of the problems that a centralised sequencer introduces, which in theory should allow them to scale better, at the cost of offering slightly relaxed semantics like a causal log order, asynchronous appends and lack of strong operations.

4 Proposed Solution

From the comparison in Section 3.3, we believe that there is an interesting opportunity to explore in the area of asynchronous systems: in theory, these systems are able to scale better when compared to synchronous systems, but often provide weak semantics that are not developer-friendly. The system we intend to implement will try to merge the benefits of both.

We will use the Eunomia system as a starting point when describing our system, and we will introduce the necessary modifications as we go along.

4.1 Overview

As stated in Section 2 our goal is to build a log based key value store that offers causal consistency and has support for ACID transactions.

Figure 2 shows a simplified view of the architecture of the system we intend to build. It will have four main components: the client, the partitions, the Eunomia service and the transaction committers. Client, partitions and the Eunomia service will have the same functionality that they have in Eunomia: clients are responsible for making requests to the partitions, the partitions will be responsible for storing the key value pairs and respond to client requests, and the Eunomia service will be responsible for producing a causally ordered log of all the events that happen in the system. The transaction committer will consume the log that Eunomia produces and use it to commit the transactions in it.

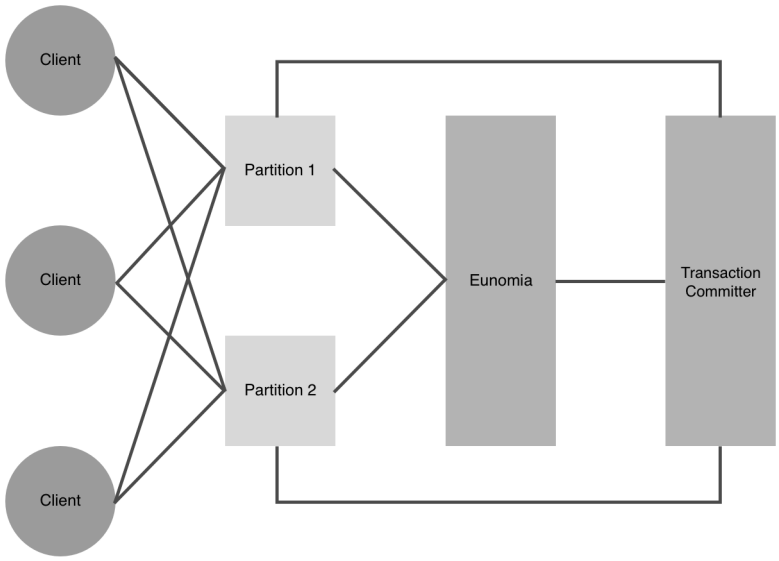

Fig. 2: System architecture of the proposed solution

Our system will ensure causal consistency by relying on the Eunomia functionality, and the support for ACID transactions will be implemented by combining the Eunomia functionality with the transaction committers. The isolation level that our transactional system will support is going to be snapshot isolation.

4.2 Transactions

In this section we will describe in detail how we are planning to implement the transactional system over the architecture we presented in the previous section. We will start by the modifications that have to be made to the Eunomia system, then we will go through the events that take part in the execution of a transaction and will finish with an example of how a transaction will work.

Architecturally the only modification that has to be made is the introduction of the transaction committer component. When it comes to metadata changes, key value pairs stored in the partitions will be tagged with what we call a version, that corresponds to the position in the log of the record that contains the update that sets that key's value, and in order to support long running read-only transactions, partitions will have to store not only one but several version for each key value pair.

We will now describe the events that take part in the execution of a transaction:

1. **Transaction Begin**: In order to start a transaction, clients will update their state to indicate that they are in the context of a transaction.
2. **Reads**: Reads made in the context of a transaction will be done from a consistent snapshot and will be recorded in the transaction read set. In order to read from a consistent snapshot the client will also keep in its state a snapshot version that will be used in every read request. The first time a client reads in the context of a transaction no snapshot version will be set, so in the first read the client will read the latest value of a certain key and will set the snapshot version equal to the version of the value he read. Subsequent reads will have the snapshot version attached and the partitions will return consistent values with that version. The snapshot version of the transaction also sets a point in time where the transaction begins.
3. **Updates**: Updates done in the context of a transaction will not be sent to the corresponding partitions. Instead they will be recorded in a buffer that will be used when the clients tries to commit the transaction.
4. **Commit Request**: In order to commit a transaction, clients will have to create a commit record and send it to one of the partitions. Partitions will then assign a timestamp to the record and will send it to Eunomia to be inserted in the log. The commit record will include the transaction snapshot, the transaction read set and the buffer with all the updates.
5. **Log Generation**: The generation of a log with all the commit records will be done using the same techniques used in Eunomia. When new commit records are received in Eunomia, they are placed in a queue of pending records, regularly Eunomia will calculate the stable timestamp and generate a log with all the records that have a stable timestamp. The generated log will then be consumed by the transaction committer.
6. **Commit/Abort Decision**: Deciding if a transaction should commit or abort is the task of the transaction committer. To do so, the transaction committer will go through every commit record in the log that is generated by Eunomia, and will check if transactions have conflicts. This will done by comparing each transaction's read set with the records in the log, from the point when the transaction started until the latest record. If a conflict is detected the transaction aborts, otherwise the transaction commits.
7. **Commit/Abort Propagation**: After deciding if a transaction commits or aborts the transaction committer will use a two-phase commit protocol to atomically commit the updates in the partitions.

Example

Consider a scenario where there are two accounts in the system, account A with its value equal to 100 and account B with its value equal to 0, and the owner of account A wants to pay the owner of account B 50 for a service. To guarantee the integrity of the operation the client will use a transaction. Figure 3 shows the inter-component communication that has to happen for this transaction to be successful.

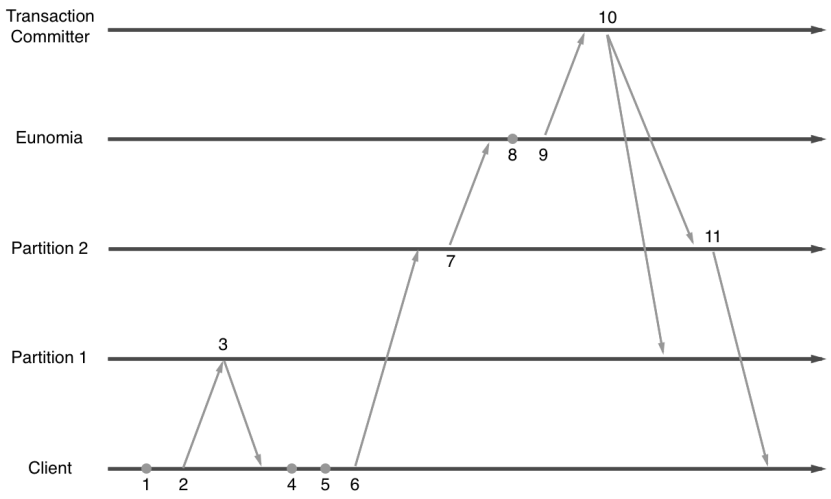

Fig. 3: Inter-component communication between components during a transaction

The client starts the operation by entering in a transaction context (1). Sends a read request to partition 1 to read the value of account A (2), to make sure account A has enough funds. Partition 1 returns the value of account A along with its version (3), that will be used as the snapshot of the transaction. Then the client makes the transfer by updating the value of account A to 50 (4) and the value of account B to 50 (5). Both updates are recorded into a buffer. Commits the transaction by sending a commit record to partition 2 (6). Partition 2 assigns the record a timestamp and sends it to Eunomia (7). Eunomia runs the stabilisation procedure and generates a log with the commit record (8). The log is propagated to the transaction committer (9) and the transaction committer commits the transaction using a two-phase commit protocol (10). For simplification reasons the two-phase commit protocol is represented in Figure 3 by a single message per partition. Finally when partition 2 receives the commit result, applies the updates of the transaction if the transaction can commit and informs the client about it (11).

4.3 Future Work

When it comes to the propagation of updates to the partitions our approach will be to use a deferred replication protocol, which means that we will only send the updates to all the partitions when the transaction commits. But there is another option that we would like to explore, which is to eagerly send the updates to the corresponding partitions and have them stay there as pending

until the transaction commits or aborts, this would improve our commit time but could potentially waste network bandwidth. In this case there are a lot of options to explore when it comes to reading the value of a key: clients read the latest committed position even if there are pending updates, read the latest pending update value optimistically, blocking until the pending updates are committed, abort the read if there are pending updates or move the transaction snapshot forward re-evaluating past reads. We will stick with the first update replication method because it is the one that is simpler to reason about, but as future work we will consider the second one by running experiments on them.

Due to the nature of transactions and their expensive cost in terms of communication we do not plan to support geo-replication. Instead we plan to support local replication of data. As a starting point our approach to replication is to delegate it to the underlying key value store used, but we plan to look further into this topic and possibly consider storing the log that Eunomia outputs as a form of data replication, like vCorfu does.

The transaction committer, just like the Eunomia service can be seen as a single point of failure. Our system will use the same techniques that Eunomia used to overcome this problem, but we plan to look further into these techniques and see if there are other alternatives that can produce better results. We also plan to investigate further the transaction committing algorithm, more specifically if there is any possibility of committing transactions in parallel by having several transaction committers consuming the Eunomia log where each one commits a subset of the transactions in the log.

5 Evaluation

As described in previous sections, our goal is to improve the scalability of log based storage systems that provide strong semantics like transactions to applications. To evaluate the validity of the proposed solution we intend to build a prototype and evaluate its implementation against real world workloads and existing systems. The evaluation will be focused on following:

- Throughput of read, write and transactional operations.
- Latency of read, write and transactional operations.
- Visibility latencies of transactional updates.
- System scalability by analysing the throughput of read, write and transactional operations as the number of clients and partitions scales.

Furthermore, we want to show the trade-offs of implementing transactions over a sequencer based architecture versus a sequencer free architecture. Additionally, if time allows, we intend to port an existing application to use our prototype and analyse the benefits of this change.

6 Scheduling of Future Work

Future work is scheduled as follows:

– January 8 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15: Deliver the MSc dissertation.

7 Conclusions

Storage systems are a fundamental building block of today's applications. This report is focused on a subclass of storage systems, the ones built over a log. We started by addressing the concept of a shared log and how it can be used as a building block of storage systems. We then surveyed the most relevant work done in the area of log based storage systems. Proposed a design of log based storage system that aims to join the benefits of an sequencer free architecture with the interfaces provided by systems that use sequencer based architectures, more specifically transactions. And concluded the report with a description of the methods and metrics we plan to use to evaluate the proposed architecture.

Acknowledgements

I am grateful to Professor Luís Rodrigues and Manuel Bravo for the fruitful discussions and comments during the preparation of this report.

References

1. Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., Davis, J.D.: Corfu: A shared log design for flash clusters. In: NSDI. (2012) 1–14
2. Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J.D., Rao, S., Zou, T., Zuck, A.: Tango: Distributed data structures over a shared log. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM (2013) 325–340
3. Wei, M., Tai, A., Rossbach, C.J., Abraham, I., Munshed, M., Dhawan, M., Stabile, J., Wieder, U., Fritchie, S., Swanson, S., et al.: vcorfu: A cloud-scale object store on a shared log. In: NSDI. (2017) 35–49
4. Tai, A., Wei, M., Freedman, M.J., Abraham, I., Malkhi, D.: Replex: A scalable, highly available multi-index data store. In: USENIX Annual Technical Conference. (2016) 337–350
5. Baker, J., Bond, C., Corbett, J.C., Furman, J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR. Volume 11. (2011) 223–234
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) **26**(2) (2008) 4

7. Nawab, F., Arora, V., Agrawal, D., El Abbadi, A.: Chariots: A scalable shared log for data management in multi-datacenter cloud environments. In: EDBT. (2015) 13–24
8. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB. (2011) 1–7
9. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference. Volume 8., Boston, MA, USA (2010) 9
10. Gunawardhana, C., Bravo, M., Rodrigues, L.: Unobtrusive deferred update stabilization for efficient geo-replication. arXiv preprint arXiv:1702.01786 (2017)