Chariots : A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments

Faisal Nawab Vaibhav Arora Divyakant Agrawal Amr El Abbadi

Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106
{nawab,vaibhavarora,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Web-based applications face unprecedented workloads demanding the processing of a large number of events reaching to the millions per second. That is why developers are increasingly relying on *scalable cloud platforms* to implement cloud applications. Chariots exposes a shared log to be used by cloud applications. The log is essential for many tasks like bookkeeping, recovery, and debugging. Logs offer linearizability and simple append and read operations of immutable records to facilitate building complex systems like stream processors and transaction managers. As a cloud platform, Chariots offers fault-tolerance, persistence, and high-availability, transparently.

Current shared log infrastructures suffer from the bottleneck of serializing log records through a centralized server which limits the throughput to that of a single machine. We propose a novel distributed log store, called the *Fractal Log Store* (FLStore), that overcomes the bottleneck of a single-point of contention. FLStore maintains the log within the datacenter. We also propose Chariots, which provides multi-datacenter replication for shared logs. In it, FLStore is leveraged as the log store. Chariots maintains causal ordering of records in the log and has a scalable design that allows elastic expansion of resources.

1. INTRODUCTION

The explosive growth of web applications and the need to support millions of users make the process of developing web applications difficult. These applications need to support this increasing demand and in the same time they need to satisfy many requirements. Fault-tolerance, availability, and a low response time are some of these requirements. It is overwhelming for the developer to be responsible for ensuring all these requirements while scaling the application to millions of users. The process is error-prone and wastes a lot of efforts by reinventing the wheel for every application.

The cloud model of computing encourages the existence of unified platforms to provide an infrastructure that provides the guarantees needed by applications. Nowadays, such an infrastructure for compute and storage services is commonplace. For example, an application can request a key-value store service in the cloud. The store exposes an interface to the client and hides all the complexities required for its scalability and fault-tolerance. We envision that a variety of programming platforms will coexist in the cloud for the developer to utilize. A developer can use multiple platforms simultaneously according to the application's needs. A micro-blogging application for example might use a key-value store platform to persist the blogs and in the same time use a distributed processing platform to analyze the stream of blogs. The shared log, as we argue next, is an essential cloud platform in the developer's arsenal.

Manipulation of shared state by distributed applications is an error-prone process. It has been identified that using immutable state rather than directly modifying shared data can help alleviate some of the problems of distributed programming [1–3]. The *shared log* offers a way to share immutable state and accumulate changes to data, making it a suitable framework for cloud application development. Additionally, the shared log abstraction is familiar to developers. A simple interface of append and read operations can be utilized to build complex solutions. These characteristics allow the development of a wide-range of applications. Solutions that provide transactions, analytics, and stream processing can be easily built over a shared log. Implementing these tasks on a shared log makes reasoning about their correctness and behavior easier and rid the developer from thinking about scalability and fault-tolerance. Also, the log provides a trace of all application events providing a natural framework for tasks like debugging, auditing, checkpointing, and time travel. This inspired a lot of work in the literature to utilize shared logs for building systems such as transaction managers, geo-replicated key-value stores, and others [6, 11, 13, 27, 28, 30, 33].

Although appealing as a platform for diverse programming applications, shared log systems suffer from a single-point of contention problem. Assigning a log position to a record in the shared log must satisfy the uniqueness and order of each log position and consequent records should have no gaps in between. Many shared log solutions tackle this problem and try to increase the append throughput of the log by minimizing the amount of work done to append a record. The most notable work in this area is the CORFU protocol [7] built on flash chips that is used by Tango [8]. The CORFU protocol is driven by the clients and uses a *centralized sequencer* that assigns offsets to clients to be filled later. This takes the sequencer out of the data path and allows the append throughput to be more than a single machine's I/O bandwidth. However, it is still limited by the bandwidth of the sequencer. This bandwidth is suitable for small clusters but cannot be used to handle larger demands encountered by large-scale web applications.

We propose **FLStore**, a distributed deterministic shared log system that scales beyond the limitations of a single machine. FL-

Store consists of a group of *log maintainers* that mutually handle exclusive ranges of the log. Disjoint ranges of the log are handled independently by different log maintainers. FLStore ensures that all these tasks are independent by using a deterministic approach that assigns log positions to records as they are received by log maintainers. It removes the need for a centralized sequencer by avoiding log position pre-assignment. Rather, FLStore adopts a *post-assignment* approach where records are assigned log positions *after* they are received by the Log maintainers. FLStore handles the challenges that arise from this scheme. The first challenge is the existence of gaps in the log that occur when a log maintainer has advanced farther compared to other log maintainers. Another challenge is maintaining explicit order dependencies requested by the application developer.

Cloud applications are increasingly employing geo-replication to achieve higher availability and fault-tolerance. Records are generated at multiple datacenters and are potentially processed at multiple locations. This is true for applications that operate on shared data and need communication to other datacenters to make decisions. In addition, some applications process streams coming from different locations. An example is Google's Photon [4] which joins streams of clicks from different datacenters. Performing analytics also requires access to the data generated at multiple datacenters. Geo-replication poses additional challenges such as maintaining *exactly-once* semantics (ensure that an event is not processed more than once), automatic datacenter-level fault-tolerance, and handling un-ordered streams.

Chariots supports multiple datacenters by providing a global replicated shared log that contains all the records generated by all datacenters. The order of records in the log must be consistent. The ideal case is to have an identical order at all datacenters. However, it is shown by the CAP theorem [12, 16] that such a **consistency** guarantee *cannot* be achieved if we are to preserve **availability** and **partition-tolerance**. In this work we favor availability and partition-tolerance, as did many other works in different contexts [14, 15, 20, 23]. Here, we relax the guarantees on the order of records in the log. In relaxing the consistency guarantee, we seek the strongest guarantee that will allow us to preserve availability and partition-tolerance. We find, as other systems have [5, 10, 19, 23, 31], that causality [21] is a sufficiently strong guarantee fitting our criterion [24].

In this paper we propose a cloud platform that exposes a shared log to applications. This shared log is replicated to multiple datacenters for availability and fault tolerance. The objective of the platform's design is to achieve high performance and scalability by allowing seamless elasticity. Challenges in building such a platform are tackled, including handling component and whole datacenter failures, garbage collection, and gaps in the logs. We motivate the log as a framework for building cloud applications by designing three applications on top of the shared log platform. These applications are: (1) a key-value store that preserves causality across datacenters, (2) a stream processing applications that handles streams coming from multiple datacenters, and (3) a replicated data store that provides strongly consistent transactions [27].

The contributions of the paper are the following:

- A design of a scalable distributed *log storage*, FLStore, that overcomes the bottleneck of a single machine. This is done by adopting a post-assignment approach to assigning log positions.
- Chariots tackles the problem of scaling causally-ordered geo-replicated shared logs by incorporating a distributed log storage solution for each replica. An elastic design is built to allow scaling to datacenter-scale computation. This is the first work we are aware of that tackles the problem of *scaling geo-replicated shared logs through partitioning*.

Consistency	Partitioned	Replicated	systems
Strong	✓	✗	CORFU/Tango [7, 8] LogBase [33] RAMCloud [29] Blizzard [25] Ivy [26] Zebra [18] Hyder [11]
Strong	✗	✓	Megastore [6] Paxos-CP [30]
Causal	✗	✓	Message Futures [27] PRACTI [10] Bayou [32] Lazy Replication [19] Replicated Dictionary [36]
Causal	✓	✓	Chariots

Table 1: Comparison of different shared log services based on consistency guarantees, support of per-replica partitioning, and replication.

The paper proceeds as the following. We first present related work in Section 2. The system model and log interface follows in Section 3. We then present a set of use cases of Chariots in Section 4. These are data management and analytics applications. The detailed design of the log is then proposed in Sections 5 and 6. Results of the evaluations are provided in Section 7. We conclude in Section 8.

2. RELATED WORK

In this paper we propose a geo-replicated shared log service for data management called Chariots. Here we briefly survey related work. We focus on systems that manage shared logs. There exist an enormous amount of work on general distributed (partitioned) storage and geo-replication. Our focus on work tackling shared logs stems from the unique challenges that shared logs pose compared to general distributed storage and geo-replication. We provide a summary of shared log services for data management application in Table 1. In the remainder of this section we provide more details about these systems in addition to other related work that do not necessarily provide log interfaces. We conclude with a discussion of the comparison provided in Table 1.

2.1 Partitioned shared logs

Several systems explored extending shared logs as a distributed storage spanning multiple machines. Hyder [11] builds a multi-version log-structured database on a distributed shared log storage. A transaction executes optimistically on a snapshot of the database and broadcasts the record of changes to *all* servers and appends a record of changes to the distributed shared log. The servers then commit the transaction by looking for conflicts in the shared log in an intelligible manner. LogBase [33], which is similar to network filesystems like BlueSky [34], and RAMCloud [29] are also multi-version log-structured databases.

Corfu [7], used by Tango [8], attempts to increase the throughput of shared log storage by employing a sequencer. The sequencer's main function is to pre-assign log position ids for clients wishing to append to the log. This increases throughput by allowing more concurrency. However, the sequencer is still a bottleneck limiting the scalability of the system.

Distributed and networked filesystems also employ logs to share their state. Blizzard [25] proposes a shared log to expose a cloud block storage. Blizzard decouples ordering and durability requirements, which improves its performance. Ivy [26] is a distributed file system. A log is dedicated to each participant and is placed in a distributed hash table. Finding data requires consulting all logs but appending is done to the participant's log only. The Zebra file system [18] employs log striping across machines to increase throughput.

2.2 Replicated shared logs

Causal replication. Causal consistency for availability is utilized by various systems [10, 19, 19, 23, 31, 32]. Recently, COPS [23] proposes causal+ consistency that adds convergence as a requirement in addition to causal consistency. COPS design aims to increase the throughput of the system for geo-replicated environments. At each datacenter, data is partitioned among many machines to increase throughput. Chariots targets achieving high throughput similarly by scaling out. Chariots differs in that it exposes a log rather than a key-value store, which brings new design challenges. Logs have been utilized by various replication systems for data storage and communication. PRACTI [10] is a replication manager that provides partial replication, arbitrary consistency, and topology independence. Logs are used to exchange updates and invalidation information to ensure the storage maintains a causally consistent snapshot. Bayou [32] is similar to PRACTI. In Bayou, batches of updates are propagated between replicas. These batches have a start and end times. When a batch is received, the state rolls back to the start time, incorporate the batch, and then roll forward the existing batches that follows. Replicated Dictionary [36] replicates a log and maintains causal relations. It allows transitive log shipping and maintains information about the knowledge of other replicas. Lazy Replication [19] also maintains a log of updates ordered by their causal relations. The extent of knowledge of other replicas is also maintained.

Geo-replicated logs. Geo-replication of a shared log has been explored by few data management solutions. Google megastore [6] is a multi-datacenter transaction manager. Megastore commit transactions by contending for log positions using Paxos [22]. Paxos-CP [30] use the log in a similar way to megastore with some refinements to allow better performance. These two systems however, operate on a serial log. All clients contend to write to the head of the log, making it a single point of contention, which limits throughput. Message Futures [27] and Helios [28] are commit protocols for strongly consistent transactions on geo-replicated data. They build their transaction managers on top of a causally-ordered replicated log that is inspired from Replicated Dictionary [36].

2.3 Summary and comparison

The related works above that build shared logs for data management applications are summarized in Table 1. We display whether the system support partitioning and replication in addition to the guaranteed ordering consistency. Consistency is either strong, meaning that the order is either identical or serializable for replicated logs or totally ordered for non-replicated logs. A system is *partitioned* if the shared log spans more than one machine for each replica. Thus, if a system of five replicas consists of five machines,

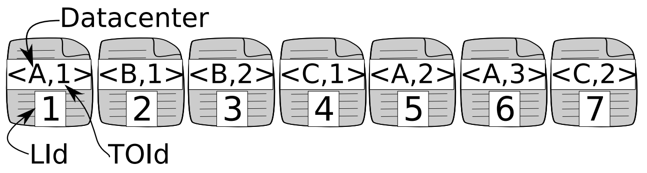

Figure 1: Records in a shared log showing their TOId inside the records alongside the datacenters that created them and the records LIds under the log

they are *not* partitioned. A system is *replicated* if the shared log has more than one *independent* copy.

Other than Chariots, the table lists four systems that support partitioning. It is possible for these systems to employ replication in the storage level. However, a blind augmentation of a replication solution will be inefficient. This is because a general-purpose replication method will have guarantees stronger than what is needed to replicate a log. The other solutions, that support replication, do not support partitioning. Handling a replica with a single node limits the achievable throughput. The processing power, I/O, and communication of a single machine can not handle the requirements of todays web applications. This is specially true for geo-replication that handles datacenter-scale demand.

Chariots attempts to fill this void of shared logs that have both a native support of replication and per-replica partitioning. This need for both replication and partitioning has been explored for different applications and guarantees, including causal consistency, *i.e.*, COPS [23]. However, geo-replication of a distributed shared log and immutable updating pose unique challenges that are not faced by geo-replication of key-value stores and block-based storage. The paper studies these challenges and design Chariots as a causally-ordered shared log that supports per-replica partitioned log storage and geo-replication.

3. SYSTEM AND PROGRAMMING MODEL

Chariots is a shared log system for cloud applications. The inner workings of Chariots are not exposed to the application developer. Rather, the developer interacts with Chariots via a set of APIs. In this section, we will show the interface used by developers to write applications using Chariots.

System model. Chariots exposes a log of records to applications. The log is maintained by a group of machines called the *log maintainers*. Collectively, these log maintainers persist a single shared log. Each log maintainer is responsible for a disjoint range of the shared log. The shared log is accessed by cloud applications, called *application clients*, through a linked library that manages the exchange of information between the application and the log maintainers. Application clients are distributed and independent from one another. And they share a single view of the shared log. The shared log is fully replicated to a number of datacenters. In our model, we adopt a view of the *datacenter as a computer* [9], an increasingly popular view of computing that reflects the demand of datacenter-scale applications.

Meta information about log maintainers, other datacenters, and the shared log are managed by *meta servers*. Meta servers are a highly-available collection of stateless servers acting as an oracle for application clients to report about the state and locations of the Log maintainers and other datacenters. This model of scale-out distributed computing and centralized stateless highly-available control servers has been shown to perform the best for large-scale systems [17].

Data model. The state of the shared log consists of the records it contains. These records are either *local copies*, meaning that they were generated by application clients residing in the same datacenter, or *external copies*, meaning that they were generated at other datacenters. Each record has an identical copy at each datacenter, one of which is considered a local copy and the other copies are considered external copies. The record consists of the contents appended by the Application client, called the record's *body*, and other *meta-information* that are used by Application clients to facilitate future access to it. The following are the meta-information maintained for each record:

- *Log Id (LId):* This id reflects the position of the record in the datacenter where the copy resides. A record has multiple copies, one at each datacenter. Each copy has a different LId that reflects its position in the datacenter's shared log.
- *Total order Id (TOId):* This id reflects the total order of the record with respect to its host datacenter, where the Application client that created it resides. Thus, copies of the same record have an identical TOId.
- *Tags:* The Application client might choose to attach tags to the record. A tag consists of a key and a value. These tags are accessible by Chariots, unlike the record's body which is opaque to the system. Records will be indexed using these tags.

To highlight the difference between LId and TOId, observe the sample shared log in Figure 1. It displays seven records with their LIds in the bottom of each record at datacenter A. The TOId is shown inside the record via the representation $< X, i >$, where X is the host datacenter of the Application client that appended the record and i is the TOId. Each record has a LId that reflects its position in the shared log of datacenter A. Additionally, each record has a TOId that reflects its order compared to records coming from the same datacenter only.

Programming interface and model. Application clients can observe and change the state of the shared log through a simple interface of two basic operations: reading and appending records. These operations are performed via an API provided by a linked software library at the Application client. The library needs the information of the meta servers only to initiate the session. Once the session is ready, the application client may use the following library calls:

1. Append(in: record, tags): Insert record to the shared log with the desired tags. The assigned TOId and LId will be sent back to the Application client. Appended records are automatically replicated to other replicas.
2. Read(in: rules, out: records): Return the records that matches the input rules. A rule might involve TOIds, LIds, and tags information.

Log records are immutable, meaning that once a record is added, it cannot be modified. If an application client desire to alter the effect of a record it can do so by appending another record that exemplifies the desired change. This principle of accumulation of changes, represented by immutable data, is identified to reduce the problems arising from distributed programming [1–3]. Taking this principled approach and combining it with the simple interface of appends and reads allows the construction of complex software while reducing the risks of distributed programming. We showcase the potential of this simple programming model by constructing data management systems in the next section.

Causality and log order. The shared log at each datacenter consists of a collection of records added by application clients at different datacenters. Ordering the records by causality relations allows sufficient consistency while preserving availability and fault-tolerance [12, 16]. Causality enforces two types of order relations [21] between read and append operations, where $o_i \rightarrow o_j$ denotes that o_i has a causal relation to o_j. A causal relation, $o_i \rightarrow o_j$, exists in the following cases:

- **Total order** for records generated from the same datacenter. If two appended records, o_i and o_j, were generated by application clients residing in the same datacenter A, then if o_i is ordered before o_j in A, then this order must be honored at all other datacenters.
- **Happened-before relations** between read and append operations. A happened-before relation exists between an append operation, o_i, and a read operation, o_j, if o_j reads the record appended by o_i.
- **Transitivity:** causal relations are transitive. If a record o_k exists such that $o_i \rightarrow o_k$ and $o_k \rightarrow o_j$ then $o_i \rightarrow o_j$.

4. CASE STUDIES

The simple log interface was shown to enable building complex data management systems [6, 11, 13, 27, 30, 33]. These systems, however, operate on a serial log with pre-assigned log positions. These two characteristics, as we argued earlier, limits the log's availability and scalability. In this section, we demonstrate data management systems that are built on top of Chariots, a causally ordered log with post-assigned log positions. The first system, *Hyksos*, is a replicated key-value store that provides causal consistency with a facility to perform *get transactions*. The second system is a stream processor that operates on streams originating from multiple datacenters. We also refer to our earlier work, Message Futures [27] and Helios [28], which provide strongly consistent transactions on top of a causally ordered replicated log. Although they were introduced with a single machine per replica implementation, their design can be extended to be deployed on the scalable Chariots.

4.1 Hyksos: causally consistent key-value store

Hyksos is a key-value store built using Chariots to provide causal consistency [21]. Put and Get operations[1] are provided by Hyksos in addition to a facility to perform get transactions (GET_TXN) of multiple keys. Get transactions return a consistent state snapshot of the read keys.

4.1.1 *Design and algorithms*

Chariots manages the shared log and exposes a read and append interface to application clients, which are the drivers of the key-value store operations. Each datacenter runs an instance of Chariots. An instance of Chariots is comprised of a number of machines. Some of the machines are dedicated to store the shared log and others are used to deploy Chariots.

The value of keys reside in the shared log. A record holds one, or more put operation information. The order in the log reflects the causal order of put operations. Thus, the current value of a key, k, is in the record with the highest log position containing a put operation. The get and put operations are performed as follows:

[1]The terms "read" and "append" are used for operations on the log and "put" and "get" are used for operations on the key-value store.

Algorithm 1 Performing Get_transactions in Hyksos

```
1: // Request the head of the log position id
2: i = get_head_of_log()
3: // Read each item in the read set
4: for each k in read-set
5:     t = Read ({tag: k, LId<i}, most-recent)
6:     Output.add (t)
```

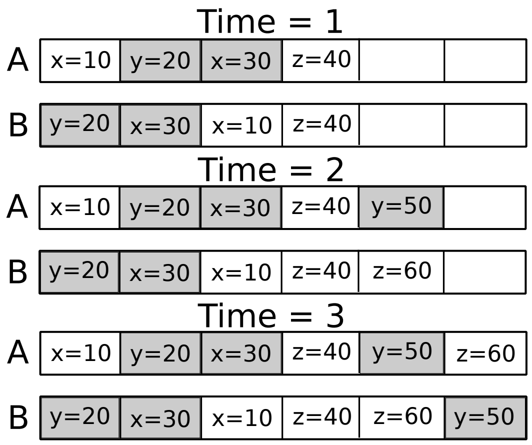

Figure 2: An example of Hyksos, the key-value store built using Chariots.

- Get(x): Perform a Read operation on the log. The read returns a recent record containing a put operation to x.

- Put(x, value): Putting a value is done by performing an Append operation with the new value of x. The record must be tagged with the key and value information to enable an efficient get operation.

Get transactions. Hyksos provides a facility to perform get transactions. The get_transaction operation returns a consistent view of the key-value store. The application client performs the Get operations as shown in Algorithm 1. First, Chariots is polled to get the head of the log's position id, i, to act as the log position when the consistent view will be taken (Line 2). There must be no gaps at any records prior to the log id. Afterwards, the application client begins reading each key k (Lines 4-6). A request to read the version of k at a log position j that satisfies the following: Record j contains the most recent write to k that is at a position less than i.

4.1.2 Example scenario

To demonstrate how Hyksos works, consider the scenario shown in Figure 2. It displays the shared logs of two datacenters, A and B. The shared log contains records of put operations. The put operation is in the form "$x = v$", where x is the key and v is the value. Records that are created by Application clients at A are shaded. Other records are created by application clients at B.

The scenario starts with four records, where each record has two copies, one at each datacenter. Two of these records are put operations to key x. The other two operations are a put to y and a put to z. The two puts to x were created at different datacenters. Note that the order of writes to x is different at A and B. This is permissible if no causal dependencies exist between them. At time 1, a Get of x at A will return 30, while 10 will be returned if the Get is performed at B.

At time 2, two Application clients, one at each datacenter, perform put operations. At A, Put(y,50) is appended to the log. At B, Put(z,60) is appended to the log. Now consider a get transaction that requests to get the value of x, y and z. First, a non-empty log position is chosen. Assume that the log position 4 is chosen. If the get transaction ran at A, it will return a snapshot of the view of the log up to log position 4. This will yield $x = 30$, $y = 20$, and $z = 40$. Note that although a more recent y value is available, it was not returned by the get transactions because it is not part of the view of records up to position 4. If the get transaction ran at B, it will return $x = 10$, $y = 20$, and $z = 40$.

Time 3 in the figure shows the result of the propagation of records between A and B. $Put(y,50)$ has a copy now at B, and $Put(z,60)$ has a copy at A.

4.2 Event processing

Another application targeted by Chariots is multi-datacenter event processing. Many applications generate a large footprint that they would like to process. The users' interactions and actions in a web application can be analyzed to generate business knowledge. These events range from click events to the duration spent in each page. Additionally, social networks exhibit more complex analytics of events related to user-generated contents (*e.g.*, micro-blogs) and user-user relationships to these events. Frequently, such analytics are carried in multiple datacenters for fault-tolerance and locality [4, 17].

Chariots enables a simple interface for these applications to manage the replication and persistence of these analytics while preserving the required exactly-once semantics. Event processing applications consist of publishers and readers. Publishing an events is as easy as performing an append to the log. Readers then read the events from the log maintainers. An important feature of Chariots is that readers can read from different log maintainers. This will allow distributing the analysis work without the need of a centralized dispatcher that can be a single-point of contention.

4.3 Message Futures and Helios

Message Futures [27] and Helios [28] are commit protocols that provide strongly consistent transactions on geo-replicated data stores. They leverage a replicated log that guarantees causal order [36]. A single node at each datacenter, call it replica, is responsible for committing transactions and replication. Transactions consist of read and write operations and are committed optimistically. Application clients read from the data store and buffer writes. After all operations are ready, a *commit request* is sent to the closest replica. A record is appended to the log to declare the transaction t as ready to begin the commit protocol. Message Futures and Helios implement different conflict detection protocols to commit transactions. Message Futures [27] waits for other datacenters to send their histories up to the point of t's position in the log. Conflicts are detected between t and received transactions, and t commits if no conflicts are detected. Helios [28] builds on a lower-bound proof that determines the lowest possible commit latency that a strongly consistent transaction can achieve. Helios commits a transaction t by detecting conflicts with transactions in a *conflict zone* in the shared log. The conflict zone is calculated by Helios using the lower-bound numbers. If no conflicts were detected, t commits. A full description of Message Futures and Helios are available in previous publications [27, 28].

Message Futures and Helios demonstrate how a causally ordered log can be utilized to provide strongly consistent transactions on replicated data. However, the used replicated log solution [36] is rudimentary and is not suitable for today's applications. It only

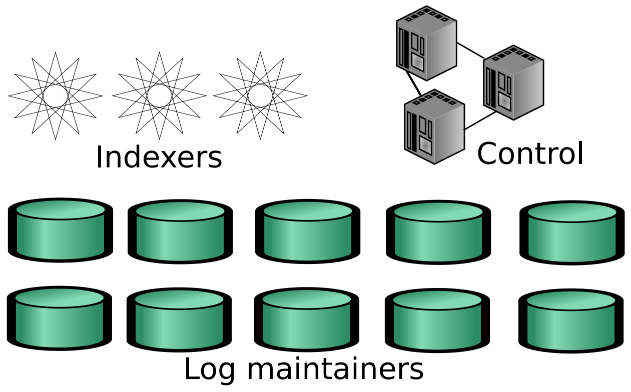

Figure 3: The architecture of FLStore

utilizes a single node per datacenter. This limits the throughput that can be achieved to that of a single node. Chariots can be leveraged to scale Message Futures and Helios to larger throughputs. Rather than a replica with a single node at each datacenter, Chariots would be used to distribute storage and computation.

4.4 Conclusion

The simple interface of Chariots enabled the design of web and analytics applications. The developer can focus on the logic of the data manager or stream processor without having to worry about the details of replication, fault-tolerance, and availability. In addition, the design of Chariots allows scalable designs of these solutions by having multiple sinks for reading and appending.

5. FLSTORE: DISTRIBUTED SHARED LOG

In this section we describe the distributed implementation of the shared log, called the Fractal Log Store (FLStore). FLStore is responsible of maintaining the log *within* the datacenter. We begin by describing the design of the distributed log storage. Then, we introduce the scalable indexing component used for accessing the shared log.

5.1 Architecture

In designing FLStore, we follow the principle of distributing computation and highly-available stateless control. This approach has been identified as the most suitable to scale out in cloud environments [17]. The architecture of FLStore consists of three types of machines, shown in Figure 3. *Log maintainers* are responsible for persisting the log's records and serving read requests. *Indexers* are responsible of access to log maintainers. Finally, control and meta-data management is the responsibility of a highly-available cluster called the *Controller*.

Application clients start their sessions by polling the Controller for information about the indexers and log maintainers. This information includes the addresses of the machines and the log ranges falling under their responsibility in addition to approximate information about the number of records in the shared log. Appends and reads are served by Log maintainers. The Application client communicates with the Controller only at the beginning of the session or if communication problems occur. And Application clients will communicate with Indexers only if read operation did not specify LIds in the rules.

5.2 Log maintainers

Scalability by post-assignment. The Log maintainers are accessed via a simple interface for adding to and reading from the shared log. They are designed to be *fully distributed* to overcome the I/O bandwidth constraints that are exhibited by current shared log protocols. A recent protocol is CORFU [7] that is limited by the I/O bandwidth of a sequencer. The sequencer is needed for CORFU to pre-assign log positions to application clients wishing to append records to the log. In FLStore, we abandon this requirement of pre-assigning log positions and settle for a *post-assignment* approach. The thesis of a post-assignment approach is to let the application client construct the record and send it to a randomly (or intelligibly) selected Log maintainer. The Log maintainer will assign the record the next available log position from log positions under its control.

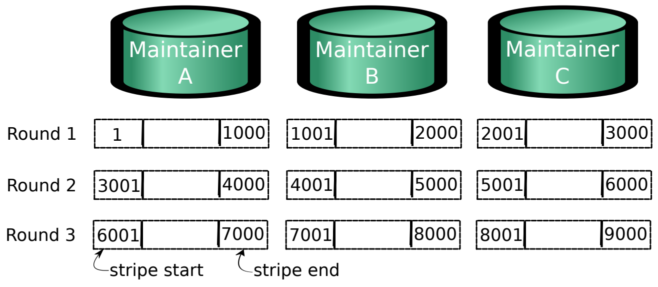

Figure 4: An example of three deterministic log maintainers with a batch size of 1000 record. Three rounds of records are shown.

Design. The shared log is distributed among the participating Log maintainers. This means that each machine holds a partial log and is responsible for its persistence and for answering requests to read its records. This distribution poses two challenges. The first is the way to append to the log while guaranteeing uniqueness and the non-existence of gaps in the log. This includes the access to these records and the way to index the records. The other challenge is maintaining explicit order guarantees requested by the application client. We employ a *deterministic* approach to make each machine responsible for specific ranges of the log. These ranges round-robin across machines where each round consists of a number of records. we will call this number the batch size. Figure 4 depicts an example of three log maintainers, *A*, *B*, and *C*. The figure shows the partial logs of the first three rounds if the batch size was set to a 1000 records.

If an application wants to read a record it directs the request to the Log maintainer responsible for it. The Log maintainer can only answer requests of records if their LIds are provided. Otherwise, the reader must collect the LIds first from the Indexers as we show in the next section. Appending to the log is done by simply sending a record or group of records to one of the Log maintainers. The Log maintainer appends the record to the next available log position. It is possible that a log maintainer will receive more record appends than others. This creates a load-balancing problem that can be solved by giving the application feedback about the rate of incoming requests at the maintainers. This feedback can be collected by the Controller and be delivered to the application clients as a part of the session initiation process. Nonetheless, this is an orthogonal problem that can be solved by existing solutions in the literature of load balancing.

5.3 Distributed indexing

Records in Log maintainers are arranged according to their LIds. However, Application clients often desire to access records according to other information. When an Application client appends a record it also *tags* it with access information. These tags depend on the application. For example, a key-value store might wish to tag a record that has Put information with the key that is written. For this reason, we utilize distributed Indexers that provide access to the Log maintainers by tag information. Distributed indexing for distributed shared logs is tackled by several systems [11, 33, 35]

Tag and lookup model. The tag is a string that describes a feature of the record. It is possible that the tag also has a value. Each record might have more than one tag. The application client can lookup a tag by its name and specify the amount of records to be returned. For example, the Application client might lookup records that has a certain tag and request returning the most recent 100 record LIds to be returned with that tag. If the tag has a value attached to it, then the Application client might lookup records with that tag and rules on the value, *e.g.*, look up records with a certain tag with values greater than *i* and return the most recent *x* records.

5.4 Challenges

Log gaps. A Log maintainer receiving more records advances in the log ahead of others. For example, Log maintainer A can have 1000 records ready while Log maintainer B has 900 records. This causes temporary gaps in the logs that can be observed by Application clients reading the log. The requirement that needs to be enforced is that *Application clients must not be allowed to read a record at log position i if there exist at least one gap at log position j less than i*.

To overcome the problem of these temporary gaps, minimal gossip is propagated between maintainers. The goal of this gossip is to identify the record LId that will guarantee that any record with a smaller LId can be read from the Log maintainers. We call this LId the *Head of the Log (HL)*. Each Log maintainer has a vector with a size equal to the number of maintainers. Each element in the vector corresponds to the maximum LId at that maintainer. Initially the vector is initialized to all zeros. Each maintainer updates its value in the local vector. Occasionally, a maintainer propagates its maximum LId to other maintainers. When the gossip message is received by a maintainer it updates the corresponding entry in the vector. A maintainer can decide that the HL value is equal to the vector entry with the smallest value. When an application wants to read or know the HL, it asks one of the maintainers for this value. This technique does not pose a significant bottleneck for throughput. This is because it is a fixed-sized gossip that is not dependent on the actual throughput of the shared log. It might, however, cause the latency to be higher as the the throughput increases. This is because of the time required to receive gossip messages and determine whether a LId has no prior gaps.

Explicit order requests. Appends translate to a total order at the datacenter after they are added by the Log maintainers. Concurrent appends therefore do not have precedence relative to each other. It is, however, possible to enforce order for concurrent appends if they were requested by the Application client. One way is to send the appends to the same maintainer in the order wanted. Maintainers ensure that a latter append will have a LId higher than ones received earlier. Otherwise, it is possible to enforce order for concurrent appends across maintainers. The Application client waits for the earlier append to be assigned a LId and then attach this LId as a minimum bound. The maintainer that receives the record with the minimum bound ensures that the record is buffered until it can be added to a partial log with LIds larger than the minimum bound. This solution however must be pursued with care to avoid a large backlog of partial logs.

6. CHARIOTS: GEO-REPLICATED LOG

In this section we show the design of Chariots that supports multi-datacenter replication of the shared log. The design is a multi-stage pipeline that includes FLStore as one of the stages. We begin the discussion by outlining an abstract design of log replication. This abstract design specifies the requirements, guarantees, and interface desired to be provided by Chariots. The abstract solution will act as a guideline in building Chariots, that will be proposed after the abstract design. Chariots is a distributed scale-out platform to manage log replication with the same guarantees and requirements of the abstract solution.

6.1 Abstract solution

Before getting to the distributed solution, it is necessary to start with an efficient **abstract solution**. This abstract solution will be provided here in the form of algorithms running on a totally ordered thread of control at the datacenter. This is similar to saying that the datacenter is the machine and it is manipulating the log according to incoming events. Using this abstract solution, we will design the distributed implementation next (Section 6.2) that will result in a behavior identical to the abstract solution with a higher performance.

The data structures used are a log and a $n \times n$ table, where n is the number of datacenters, called the Awareness Table (ATable) inspired by Replicated Dictionary [36]. The table represents the datacenter's (DC's) extent of knowledge about other DCs. Each row or column represents one DC. Consider DC A and its ATable T_A. The entry $T_A[B,C]$ contains a TOId, τ, that represents B's knowledge about C's records according to A. This means that A is certain that B knows about all records generated at host DC C up to record τ. When a record is added to the log, it is tagged by the following information: (1) *TOId*, (2) *Host datacenter Id*, and (3) causality information.

The body of the record, which is supplied by the application is opaque to Chariots . To do the actual replication, the local log and ATable are continuously being propagated to other DCs. When the log and ATable are received by another DC, the new records are incorporated at the receiving log and the ATable is updated accordingly.

The algorithms to handle operations and propagation are presented with the assumption that only one node manipulates Chariots, containing the log of records and ATable. In Section 6.2 we will build the distributed system that will be manipulating Chariots while achieving the correct behavior of the abstract solution's algorithms presented here. The following are the events that need to be handled by Chariots:

1. **Initialization**: The log is initialized and the ATable entries are set to zero. Note that the first record of each node has a TOId of 1.

2. **Append**: Construct the record by adding the following information: host identifier, TOId, LId, causality, and tags. Update the entry $T_I[I,I]$, where I is the host datacenter's id, to be equal to the record's TOId. Finally, add the record to the log.

3. **Read**: Get the record with the specified LId.

4. **Propagate**: A snapshot of Chariots is sent to another DC j. The snapshot includes a subset of the records in the log that are not already known by j. Whether a record, r, is known to j can be verified using $T_i[j,i]$ and comparing it to $TOId(r)$.

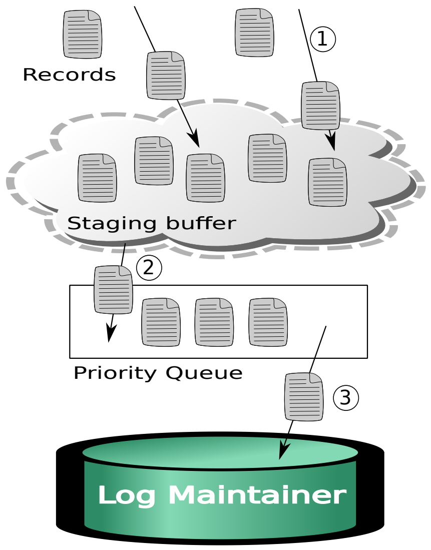

Figure 5: The components involved in adding records in the abstract solution.

5. **Reception**: When a log is received, incorporate all the records that were not seen before to the local log if its causal dependencies are satisfied. Otherwise, add the record with unsatisfied dependencies to a priority queue ordered according to causal relations. This is depicted in Figure 5. The incoming records are all put in a staging buffer (step 1) and are taken and added to the log or priority queue according to their causal dependencies (step 2). Chariots checks the priority queue frequently to transfer any records that have their dependencies satisfied to the log (step 3). Also, the ATable is updated to reflect the newly incorporated records.

Garbage collection. The user has the choice to either garbage collect log records or maintain them indefinitely. Depending on the applications, keeping the log can have great value. If the user choses not to garbage collect the records then they may employ a cold storage solution to archive older records. On the other hand, the user can choose to enable garbage collection of records. It is typical to have a temporal or spatial rule for garbage collecting the log. However, in addition to any rule set by the system designer, garbage collection is performed for records only after they are known by all other replicas. This is equivalent to saying that a record, r, can be garbage collected at i if and only if $\forall_{j \in nodes}(T_i[j, host(r)] \geq ts(r))$, where $host(r)$ is the host node of r.

6.2 Chariots distributed design

In the previous section we showed an efficient abstract design for a shared log that supports multi-datacenter replication. Chariots is a distributed system that mimics that abstract design. Each datacenter runs an instance of Chariots. The shared logs at different datacenters are replicas. All records exist in all datacenters. The system consists of a multi-stage pipeline. Each stage is responsible of performing certain tasks to incoming records and pushing them along the pipeline where they eventually persist in the shared log. Each stage in Chariots is designed to be elastic. An important design principle is that Chariots is designed to identify bottlenecks in the pipeline and allow overcoming them by adding more resources to the stages that are overwhelmed. For this to be successful, elasticity of each stage is key. Minimum to no dependencies exist between the machines belonging to one stage.

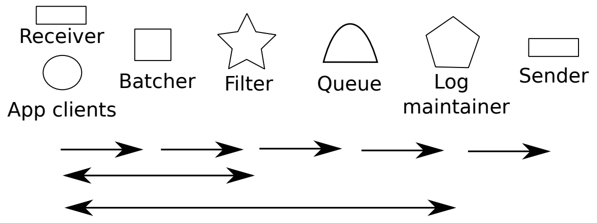

Figure 6: The components of the multi-data center shared log. Arrows denote communication pathways in the pipeline.

Pipeline design. Chariots pipeline consists of six stages depicted in Figure 6. The first stage contains nodes that are generating records. These are Application clients and machines receiving the records sent from other datacenters. These records are sent to the next stage in the pipeline, *Batchers*, to batch records to be sent collectively to the next stage. *Filters* receive the batches and ensure the uniqueness of records. Records are then forwarded to Queues where they are assigned LId. After assigning a LId to a record it is forwarded to FLStore that constitutes the Log maintainers stage. The local records in the log are read from FLStore and sent to other datacenters via the *Senders*.

The arrows in Figure 6 represent the flow of records. Generally, records are passed from one stage to the next. However, there is an exception. Application clients can request to read records from the Log maintainers. Chariots support elastic expansion of each stage to accommodate increasing demand. Thus, each stage can consist of more than one machine, *e.g.*, five machines acting as Queues and four acting as Batchers. The following is a description of each stage:

Application clients. The Application client hosts the application modules. These modules uses the interface to the log that was presented in Section 3. Some of the commands are served by only reading the log. These include Read and control commands. These requests are sent directly to the Log maintainers. The Append operation creates a record that encapsulates the user's data and send it to any Batcher machine.

Batchers. The Batchers buffer records that are received locally or from external sources. Batchers are completely independent from each other, meaning that no communication is needed from one Batcher to another and that scaling to more batchers will have no overhead. Each Batcher has a number of buffers equal to the number of Filters. Each record is mapped to a specific Filter to be sent to it eventually. Once a buffer size exceeds a threshold, the records are sent to the designated Filter. The way records are mapped to Filters is shown next.

Filters. The Filters ensures uniqueness of records. To perform this task, each Filter becomes a champion for a subset of the records. One natural way to do so is to make each Filter a champion for records with the same host Id, i.e. the records that were created at the same datacenter. If the number of Filters needed is less that the number of datacenters, then a single Filter can be responsible for more than one datacenter. Otherwise, if the number of needed Filters is in fact larger than the number of datacenters, then more than one Filter need to be responsible for a single datacenter's records. For example, consider that two Filters, x and y, responsible for records coming from datacenter A. Each one can be responsible for a subset of the records coming from A. This can be achieved by leveraging the unique, monotonically increasing, TOIds. Thus, x can be responsible for ensuring uniqueness of A's records with odd

TOIds and *y* can ensure the uniqueness of records with even TOIds. Of course, any suitable mapping can be used for this purpose. To ensure uniqueness, the processing agent maintains a counter of the next expected TOId. When the next expected record arrives it is added to the batch to be sent to the one of the Queues. Note also that this stage does not require any communication between filters, thus allowing seamless scalability.

Queues. Queues are responsible for assigning LIds to the records. This assignment must preserve the ordering guarantees of records. To append records to the shared log they need to have all their causal dependencies satisfied in addition to the total order of records coming from the same datacenter. Once a group of records have their causal dependencies satisfied, they are assigned LIds and sent to the appropriate log maintainer for persistence. For multi-datacenter records with causal dependencies, it is not possible to append to the FLStore directly and make it assign LIds in the same manner as the single-datacenter deployment shown in section 5.2. This is because it is not guaranteed that any record can be added to the log at any point in time, rather, its dependencies must be satisfied first. The queues ensure that these dependencies are preserved and assign LIds for the records before they are sent to the log maintainers. The queues are aware of the deterministic assignment of LIds in the log maintainers and forward the records to the appropriate maintainer accordingly.

Queues ensure causality of LId assignments by the use of a token. The token consists of the current maximum TOId of each datacenter in the local log, the LId of the most recent record, and the deferred records with unsatisfied dependencies. The token is initially placed at one of the Queues. The Queue holding the token append all the records that can be added to the log. The Queue can verify whether a record can be added to the shared log by examining the maximum TOIds in the token. The records that can be added are assigned LIds and sent to the Maintainers designated for them. The token is updated to reflect the records appended in the log. Then, the token is sent to the next maintainer in a round-robin fashion. The token might include all, some, or none of the records that were not successfully added to the log. Including more deferred records with the token consumes more network I/O. On the other hand, not forwarding the deferred records with the token might increase the latency of appends. It is a design decision that depends on the nature of Chariots deployment.

Log maintainers. These Log maintainers are identical to the distributed shared log maintainers of FLStore presented in Section 5.2. Maintainers ensure the persistence of records in the shared log. The record is available to be read by senders and application clients when they are persisted in the maintainers.

Log propagation. Senders propagate the local records of the log to other datacenters. Each sender is limited by the I/O bandwidth of its network interface. To enable higher throughputs, more Senders are needed at each datacenter. Likewise, more Receivers are needed to receive the amount of records sent. Each Sender machine is responsible to send parts of the log from some of the maintainers to a number of Receivers at other datacenters.

6.3 Live elasticity

The demand on web and cloud applications vary from time to time. The ability of the infrastructure to scale to the demand seamlessly is a key feature for its success. Here, we show how adding compute resources to Chariots in the fly is possible without disruptions to the Application clients. The elasticity model of Chariots is to treat each stage as an independent unit. This means that it is possible to add resources to a single stage to increase its capacity without affecting the operation of other stages.

Completely independent stages. Increasing the capacity of completely independent stages merely involves adding the new resources and sending the information of the new machine to the higher layer. The completely independent stages are the receivers, batchers, and senders. For adding a receiver, the administrator needs to inform senders of other datacenters so that it can be utilized. Similarly, a new batcher need to inform local receivers of its existence. A new sender is different in that it is the one reading from log maintainers, thus, the log maintainers need not be explicitly told about the introduction of a new sender.

Filters. In Chariots, each filter is championing a specific subset of the log records. Increasing the number of filters results in the need of reassigning championing roles. For example, a filter that was originally championing records from another datacenter could turn out to be responsible for only a subset of these records while handing off the responsibility of the rest of them to the new filter. This reassignment need to be orchestrated with batchers. There need to be a way for batchers to figure out when the hand-over took place so that they can direct their records accordingly. A *future reassignment* technique will be followed for filters as well as log maintainers as we show next. A future reassignment for filters begin by marking future TOIds that are championed by the original filter. These future TOIds mark transition of championing a subset of the records to the new filter. Consider a filter that champions records from datacenter *A* in a reassignment scenario of adding a new filter that will champion the subset of these records with even TOIds. Marking a future TOId, *t*, will result in records with even TOIds greater than *t* to be championed by the new filter. This future reassignment should allow enough time to propagate this information to batchers.

Queues. Adding a new queue involves two tasks: making the new queue part of the token exchange loop and propagating the information of its addition to filters. The first task is performed by informing one of the queues that it should forward the token to the new queue rather than the original neighbor. The latter task (informing filters) can be performed without coordination because a queue can receive any record.

Log maintainers. Expanding log maintainers is similar to expanding filters in that each maintainer champions a specific set of records. In this case, each log maintainer champions a subset of records with specific LIds. The future reassignment technique is used in a similar way to expanding filters. In this case, not only do the queues need to know about the reassignment, but the readers need to know about it too. Another issue is that log maintainers persist old records. Rather than migrating the old records to the new champion, it is possible to maintain an epoch journal that denotes the changes in log maintainer assignments. These can be used by readers to figure out which log maintainer to ask for an old record.

7. EVALUATION

In this section we present some experiments to evaluate our implementation of FLStore and Chariots. The experiments were conducted on a cluster with nodes with the following specifications. Intel Xeon E5620 CPUs that are running 64-bit CentOS Linux with OpenJDK 1.7 were used. The nodes in a single rack are connected by a 10GB switch with an average RTT of 0.15 ms. We also perform the baseline experiments on Amazon AWS. There, we used compute optimized machines (c3.large) in the datacenter in Virginia. Each machine has 2 virtual CPUs and a 3.75 GiB memory. We refer to the earlier setup as the *private cloud* and the latter setup as the *public cloud*. Unless it is mentioned otherwise, the size of each record is 512 Bytes.

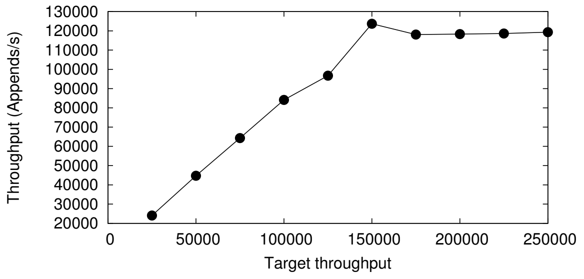

Figure 7: The throughput or one maintainer while increasing the load in a public cloud

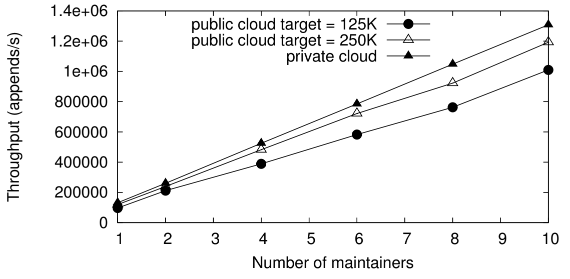

Figure 8: The append throughput of the shared log in a single-datacenter deployment while increasing the number of Log Maintainers.

7.1 FLStore scalability

The first set of experiments that we will present is of the FLStore implementation which operates within the datacenter. Each one of the experiments consists of two types of machines, namely Log maintainers and clients. The clients generate records and send them to the Log Maintainers to be appended. We are interested in measuring the scaling behavior of FLStore while increasing the number of maintainers. We begin by getting a sense of the capacity of the machines. Figure 8 shows the throughput of one maintainer in the public cloud while increasing the load on it. Records are generated with a specific rate at each experiment point from other machines. The rate is called the *target throughput*. Note how as the target throughput increases, the achieved throughput increases up to a point and then plateaus. The maximum throughput is achieved when the target throughput is 150K and then drops to be around 120K appends per second. These numbers will help us decide what target throughputs to choose for our next experiments.

To verify the scalability of FLStore, Figure 8 shows the cumulative throughput of different scenarios each with a different number of maintainers. For each experiment an identical number of client machines were used to generate records to be appended. Ideally, we would like the throughput to increase linearly with the addition of new maintainers. Three plots are shown, two from the public cloud and one from the private cloud. The ones from the public cloud differ in the target throughput to each maintainer. One targets 125K appends per second for each maintainer while the other targets 250K appends per second. Note how one is below the plateau point and one is above. The figure shows that FLStore scales with the addition of resources. A single maintainer has a throughput of 131K for the private cloud, 96.7K for the public cloud with a target of 125K, and 119K for the public cloud with the target of 250K. As we are increasing the number of Log Maintainers a near-linear scaling is observed. For ten Log Maintainers, the achieved append throughput was 1308034 record appends per second for the private cloud. This append throughput is 99.3% when compared to a perfect scaling case. The public cloud case with a target of 125K achieves a throughput that is slightly larger than the perfect scaling case. This is due to the variation in machines' performances. The other public cloud case achieve a scaling of 99.9%. This near-perfect scaling of FLStore is encouraging and demonstrates the effect of removing any dependencies between maintainers.

Machine	Throughput (Kappends/s)
Client	129
Batcher	129
Filter	129
Maintainer	124
Store	132

Table 2: The throughput of machines in a basic deployment of Chariots with one machine per stage.

Machine	Throughput (Kappends/s)
Client 1	120
Client 2	122
Batcher	126
Filter	125
Maintainer	123
Store	132

Table 3: The throughput of machines in a deployment of Chariots with two clients and one machine per stage for the remaining stages.

7.2 Chariots scalability

The full deployment of Chariots that is necessary to operate in a multi-datacenter environment consists of five stages. These stages are described in Section 6.2. Here, we will start from a basic deployment of one machine per stage in the private cloud. We observe the throughput of each stage to try to identify the bottleneck. Afterward, we observe how this bottleneck can be overcome by increasing resources. The simple deployment of one machine per stage of Chariots pipeline achieves the throughputs shown in Table 2. The table lists the throughput in Kilo records per second for each machine in the pipeline. Note how all machines achieve a similar throughput of records per second. It is possible for the store to achieve a throughput higher than the client because of the effect of buffering. Close throughput numbers for all machines indicates that the bottleneck is possibly due to the clients. The clients might be generating less records per second than what can be handled by the pipeline.

To test this hypothesis we increase the number of machines generating records to two client machines. The results are shown in Table 3. If the clients were indeed not generating enough records to saturate the pipeline, then we should observe an increase in the throughput of the Batcher. However, this was not the case. The increased load actually resulted in a lower throughput for the batcher. This means that the batcher is possibly the bottleneck. So, we increase the number of batchers to observe the throughput of latter stages in the pipeline. Table 4 shows the throughput of machines with two client machines, two batchers, and a single machine for each of the remaining stages. Both batchers achieve a throughput that is higher than the one achieved by a single batcher in the pre-

Machine	Throughput (Kappends/s)
Client 1	126
Client 2	129
Batcher 1	149
Batcher 2	129
Filter	120
Maintainer	118
Store	121

Table 4: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages.

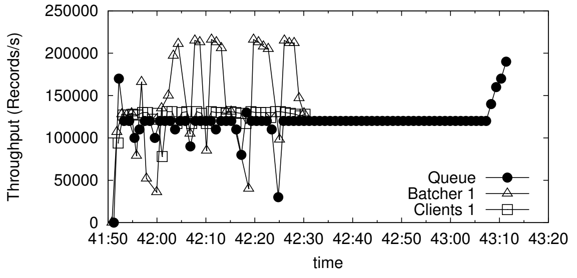

Figure 9: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages

vious experiments. This means that the throughput of the Batcher stage more than doubled. However, now the bottleneck is pushed to the filter stage that is not able to handle more than 130000 records per second. Because the throughput of latter stages is almost half the throughput of the Batcher, they take twice the time to finish the amount of records generated by the clients (10000000 records). The throughput timeseries for one client, one batcher, and the queue are shown in Figure 9. We did not show all the machines' throughputs to avoid cluttering the figure. The Batchers are done with the records at time 42:30, whereas, the latter stages lasted till time 43:10. Note that by the end of the experiment, the throughput of the queue increases abruptly. The reason for this increase is that the although the Batchers had already processed the records they are still transmitting them to the Filter until time 43:08, right before the abrupt increase. The network interface's I/O of the Filter was limiting its throughput. After it is no longer receiving from the two Batchers it can send with a higher capacity to the latter stages, thus causing an increase in the observed throughput. This is also the reason of why in the beginning of the experiment, a higher throughput is observed for some of the stages (*e.g.*, the high throughput in the beginning for the queue). The reason is that they still had capacity in their network's interface I/O before it was also used to propagate records to latter stages. Another interesting observation is the performance variation of the batcher. This turned out to be a characteristic of machines at a stage generating more throughput than what can be handled by the next stage.

Increasing the number of machines further should yield a better throughput. We experiment with the previous setting, but this time with two machines for all stages. The throughput values records are presented in Table 5. Note how all stages are scaling. The throughput of each stage has doubled. Each machine achieves a close throughput to the basic case of a pipeline with one machine per stage.

Machine	Throughput (Kappends/s)
Client 1	130
Client 2	130
Batcher 1	127
Batcher 2	127
Filter 1	127
Filter 2	126
Maintainer 1	125
Maintainer 2	126
Store 1	137
Store 2	137

Table 5: The throughput of machines in a deployment of Chariots with two machines per stage.

Our main objective is to allow scaling of shared log systems to support today's applications. We showed in this evaluation how a FLStore deployment is able to scale while increasing the number of maintainers within the datacenter (Figure 8). Also, we evaluated the full Chariots pipeline that is designed to be used for multi-datacenter environments. The bottleneck of a basic deployment was identified and Chariots overcome it by adding more resources to the pipeline.

8. CONCLUSION

In this paper we presented a shared log system called Chariots. The main contribution of Chariots is the design of a distributed shared log system that is able to scale beyond the limit of a single node. This is enabled by a deterministic post-assignment approach of assigning ranges of records to Log maintainers. Chariots also increases the level of availability and fault-tolerance by supporting geo-replication. A novel design to support a shared log across datacenters is presented. Causal order is maintained across records from different datacenters. To allow scaling such a shared log, a multi-stage pipeline is proposed. Each stage of the pipeline is designed to be scalable by minimizing the dependencies between different machines. An experimental evaluation demonstrated that Chariots is able to scale with increasing demand by adding more resources.

9. ACKNOWLEDGMENTS

This work is supported by NSF grant IIS 1018637 and is partially funded by a gift grant from NEC Labs America. Faisal Nawab is partially funded by a scholarship from King Fahd University of Petroleum and Minerals. We would also like to thank Amaozn for access to Amazon EC2.

References

[1] J. Bonér. The Road to Akka Cluster and Beyond. https://www.youtube.com/watch?v=2wSYcyWCtx4.

[2] N. Marz. How to beat the CAP theorem. http://bit.ly/marz-cap-theorem.

[3] P. Helland. Immutability changes everything! http://vimeo.com/52831373. Talk at RICON, 2012.

[4] R. Ananthanarayanan et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588. ACM, 2013.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, pages 761–772. ACM, 2013.

[6] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[7] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *NSDI*, pages 1–14, 2012.

[8] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, pages 325–340. ACM, 2013.

[9] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[10] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *NSDI*, 2006.

[11] P. Bernstein, C. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[12] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[13] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein. Edelweiss: Automatic storage reclamation for distributed programming. *Proceedings of the VLDB Endowment*, 7(6), 2014.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[17] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[18] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP*, pages 401–416. ACM, 2011.

[24] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[25] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI*, pages 257–273. USENIX, 2014.

[26] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[27] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*, 2013.

[28] F. Nawab, V. Arora, D. Agrawal, and A. E. Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *SIGMOD*, 2015.

[29] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41. ACM, 2011.

[30] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. E. Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. *Flexible update propagation for weakly consistent replication*, volume 31. ACM, 1997.

[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*. ACM, 1995.

[33] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[34] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, page 19, 2012.

[35] S. Wang, D. Maier, and B. C. Ooi. Lightweight indexing of observational data in log-structured storage. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[36] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.