Geo-Replication in Large Scale Cloud Computing Applications

(extended abstract of the MSc dissertation)

Sérgio Almeida
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

Abstract—**The work described in the thesis proposes a Geo-distributed key-value datastore, named ChainReaction, that offers *causal+* consistency, with high performance, fault-tolerance, and scalability. ChainReaction avoids the bottlenecks of linearizability while providing competitive performance when compared with systems merely offering eventual consistency. We have experimentally evaluated the benefits of our approach by applying the Yahoo! Cloud Serving Benchmark.**

I. Introduction

To manage the tradeoffs among consistency and performance, in particular for systems supporting Geo-replication, is one of the most challenging aspects in the design of datastores for cloud-computing applications. Some of these tradeoffs have been captured by the well-known CAP Theorem [1], that states that is impossible to offer simultaneously consistency, availability, and partition-tolerance. As a result, several datastores have been proposed in the last few years, implementing different combinations of consistency guarantees and replication protocols [2], [3], [4], [5], [6], [7], [8]. Some solutions opt to weaken consistency, offering only eventual consistency, in order to achieve the desired efficiency. Unfortunately, week consistency imposes a complexity burden on the application programmer. On the other hand, solutions that use stronger consistency models, such as linearizability, provide very intuitive semantics to the programmers but suffer from scalability problems.

The work developed makes a step forward in this path, by proposing a novel datastore design, named ChainReaction. Our solution relies on a specialized variant of chain-replication [9], that offers the *causal*+ consistency criteria (recently formalized in [8]) and is able to leverage from the existence of multiple replicas to distribute the load of read-requests. As a result, ChainReaction avoids the bottlenecks of linearizability while providing competitive performance when compared with systems merely offering eventual consistency. Furthermore, ChainReaction can be deployed either on a single datacenter or on Geo-replicated scenarios, over multiple datacenters. Finally, and similarly to [8] our solution also provides a transactional construct that allows a client to read the value of multiple objects in a *causal*+ consistent way.

We have experimentally evaluated the benefits of our approach by applying the Yahoo! Cloud Serving Benchmark to a prototype deployment that includes our own solution as well as Apache Cassandra [2], FAWN-KV [3], and a system that emulates COPS [8].

The remaining of this document is organized as follows. Section II addresses related work. Section III discusses the operation of ChainReaction in a single datacenter, Section IV presents our extensions to support Geo-replication, and Section V describes the support for GET-TRANSACTIONS. Section VI provides insights on the current implementation. Section VII presents the results of the experimental evaluation. Section VIII concludes the document.

II. Related Work

A datastore for Geo-replicated systems, in a cloud computing environment, must address the occurrence of faults, client locality (latency), and avoid blocking in face of network partitions. Therefore, it has to implement some form of replication, including replication over the wide-area. Ideally, such datastore would provide lineralizability [10], as this is probably the most intuitive model for programmers. Unfortunately, as the CAP theorem shows, a strongly consistent system may block (and therefore become unavailable) if a network partition occurs, something that is not unlikely in a Geo-replicated scenario. Furthermore, even if no partitions occur, strong consistency is generally expensive to support in a replicated system, because of the need to totally order all write operations. Therefore, datastores for these environments need to make some tradeoff between consistency, availability, and efficiency.

Among the most relevant techniques to implement data replication we can enumerate: active replication (typically, based on some form of Paxos [11] variant), passive replication (often, primary-backup), quorums, and chain-replication. We will not provide a description of all of these techniques since they are well understood today. We just provide a brief introduction to chain replication[9] given that this technique provides linearizability, somewhat high throughput, and availability. This approach organizes replicas in a *chain topology*. Write operations are directed to the head of the chain and are propagated until they reach the tail. At this point the tail sends a reply to the client and the write finishes. Contrary to write operations, read operations are always routed to the tail. Since all the values stored in the tail are guaranteed to have been propagated to all replicas, reads are always consistent. Chain replication exhibits a higher latency than multicast-based replication solutions but, on the other hand, it is extremely resource efficient and, therefore, it has been adopted in several practical systems. FAWN-KV [3] and Hyperdex [12] are two datastores that offer strong consistency using chain-replication as the main replication technique. CRAQ [6] is also based on chain replication but, for performance, supports eventual consistency by not constraining reads to be executed on the tail.

Apache Cassandra [2] supports Geo-replication and uses a quorum technique to maintain replicas consistent. However quorums can only be configured to offer weak consistency guarantees, namely eventual consistency. Quorums are also used by Amazon's Dynamo [4], although it only provides eventual consistency over the wide-area resorting to conflict resolution mechanisms based on vector clocks. Also the work described in [13] provides a simple adaptation of the chain replication protocol to a Geo-replicated scenario including multiple datacenters. This solution avoids intra-chain links over the wide area network. Google Megastore [14] is also deployable in a multi datacenter scenario providing serializable transactions over the wide area network, and relies on (blocking) consensus to ensure consistency.

COPS [8] is a datastore designed to provide high scalability over the wide-area. For this purpose, COPS has proposed a weak consistency model that, in opposition to eventual consistency, can provide precise guarantees to the application developer. This consistency model, named *causal+*, ensures that operations are executed in an order that respects causal order [15] and that concurrent operations are eventually ordered in a consistent way across datacenters[1]. To provide such guarantees, COPS requires clients to maintain metadata that encodes *dependencies* among operations. These dependencies are included in the write requests issued by a client. COPS also introduces a new type of operations named *get-transactions*. These operations allow a client to read a set of keys ensuring that the dependencies of all keys have been met before the values are returned.

III. Single Site ChainReaction

We now describe the operation of ChainReaction in a single site. The description of the extensions required to support Geo-replication is postponed to Section IV. We start by briefly discussing the consistency model offered by ChainReaction, followed by a general overview and, subsequently, a description of each component of the architecture.

A. Consistency Model

We have opted to offer the *causal+* consistency model [16], [17], [8]. We have selected *causal+* because it provides a good tradeoff among consistency and performance. Contrary to linearizability, *causal+* allows for a reasonable amount of parallelism in the processing of concurrent requests while still ensuring that concurrent write operations are totally ordered, thus avoiding the existence of divergent replicas (a common problem of causal consistency). On the other hand, and in opposition to eventual consistency, it provides precise guarantees about the state observed by applications. Similarly to COPS [8], our system also supports GET-TRANSACTIONS, that allows an application to obtain a *causal+* consistent snapshot of a set of objects.

[1]In fact, a similar consistency model has been used before, for instance in[16], [17] , but was only coined as *causal+* in [8].

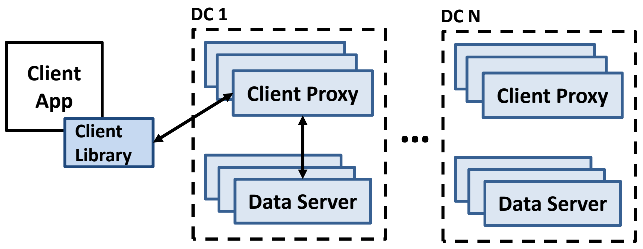


Figure 1. Overview of the ChainReaction architecture.

B. Architecture Overview

The architecture of ChainReaction is based on the FAWN-KV system [3]. We consider that each datacenter is composed of multiple *data servers* (back-ends) and multiple *client proxies* (front-ends). Data servers are responsible for serving read and write requests for one or more data items. Client proxies receive the requests from end-users (for instance a browser) or client applications and redirect the requests to the appropriate data server. An overview of ChainReaction architecture is presented in Figure 1.

Data servers self-organize in a DHT ring such that consistent hashing can be used to assign data items to data servers. Each data item is replicated across R consecutive data servers in the DHT ring. Data servers execute the chain-replication protocol to keep the copies of the data consistent: the first node in the ring serving the data item acts as head of the chain and the last node acts as tail of the chain. Note that, since consistent hashing is used, a data server may serve multiple data items, thus, being a member of multiple chains (*i.e.*, head node for one chain, tail for another, and a middle node for $R-2$ chains).

We further assume that, in each datacenter, the number of servers, although large, can be maintained in a one-hop DHT [18]. Therefore, each node in the system, including the client proxies, can always locally map keys to servers without resorting to DHT routing or to an external directory.

Considering the architecture above, we now describe the lifecycle of a typical request in the FAWN-KV system which employs a classical chain-replication solution. ChainReaction uses a variant of this workflow that will be explained in the next subsections. The client request is received by a client proxy. The proxy uses consistent hashing to select the first server to process the request: if it is a write request it is forwarded to the head data server of the corresponding chain; if it is a read request, it is forwarded directly to the tail data server. In the write case, the request is processed by the head and then propagated "down" in the chain until it reaches the tail. For both read and write operations the tail sends the reply to the proxy which, in turn, forwards an answer back to the source of the request.

C. A Chain Replication Variant

The operation of the original chain replication protocol, briefly sketched above, is able to offer linearizable executions. In fact, read and write operations are serialized at a single node, the tail of the chain. The drawback of this approach is that the existing replicas are not leveraged to promote load balancing among concurrent read operations. In ChainReaction we decided to provide *causal*+ consistency as this allows us to make a much better utilization of the resources required to provide fault-tolerance.

Our approach departs from the following observation: if a node x, in the chain, is causally consistent with respect to some client operations, then all nodes that are predecessors of x in the chain are also causally consistent. This property trivially derives from the *update invariant* of the original chain replication protocol. Therefore, assume that a node observes a value returned by node x for a given object O, as a result of a read or a write operation *op*. Future read operations over O that causally depend on *op* are constrained to read from any replica between the head of the chain and node x, in order to obtain a consistent state (according to the *causal*+ criteria). However, as soon as the operation *op* becomes stable (*i.e.*, when it reaches the tail), new read operations are no longer constrained, and a consistent state can be obtained by reading *any* server in the chain.

ChainReaction uses this insight to distribute the load of concurrent read requests among all replicas. Furthermore, it permits to extend the chain (in order to have additional replicas for load balancing) without increasing the latency of write operations, by allowing writes to return as soon as they are processed by the first k replicas (where k defines the fault-tolerance of the chain, k is usually lower than the total number of replicas). The propagation of writes from node k until the tail of the chain can be performed lazily.

To ensure the correctness of read operations according to the *causal*+ consistency model across multiple objects, clients are required to know the chain position of the node that processed its last read request for each object they have read. To store this information we use a similar strategy as the one used in COPS. We maintain *metadata* entries stored by a *client library*. However, contrary to COPS, we do not require each individual datacenter to offer linearizability as this is an impairment to scalability ([8] relies on a classical chain-replication solution to provide this). Additionally, we ensure that the results of write operations only become visible when all their causal dependencies have become stable in the local datacenter. This allows the versions divergence to be only *one level deep*, which avoids violation of the causal order when accessing multiple objects.

D. Client Interface and Library

The basic API offered by ChainReaction is similar to that of most existing distributed key-value storage systems. The operations available for clients are the following. **PUT (key, val)** that allows to assign (write) the value *val* to an object identified by *key*. **val** $\leftarrow$ **GET (key)**, that returns (reads) the value of the object identified by the *key*, reflecting the outcome of previous PUT operations.

These operations are provided by a client library that is responsible for managing client metadata, which is then automatically added to requests and extracted from replies. When considering a system deployed over a single datacenter, the metadata stored by the client library is in the form of a table, which includes one entry for each accessed object. Each entry comprises a tuple on the form (*key, version*, *chainIndex*). The *chainIndex* consists of an identifier that captures the chain position of the node that processed and replied to the last request of the client for the object to which the metadata refers. When a client makes a read operation on a data item identified by *key*, it must present the metadata above. Furthermore, ChainReaction can update the metadata as a result of executing such an operation.

E. Processing of Put Operations

We now provide a detailed description on how PUT operations are executed in ChainReaction. When a client issues a PUT operation using the Client API, the client library makes a request to a client proxy including the *key* and the value *val*. The client library tags this request with the metadata relative to the last PUT performed by that client as well as the metadata that relate to the GET operations performed over any objects since that PUT. Metadata is only maintained for objects whose version is not stable yet; stable versions do not put constraints on the execution of PUT or GET operations (we discuss GET operation further ahead). This allows to control the amount of metadata that is required to be stored at each client.

Because we aim at boosting the performance of read operations while ensuring *causal*+ consistency guarantees, we have opted to delay (slightly) the execution of PUT operations on chains, as to ensure that the version of any object from which the current PUT casually depends has become stable in its respective chain (*i.e.*, the version has been applied to the respective tail). This ensures that no client is able to read mutually inconsistent versions of two distinct objects. This is achieved using a *dependency stabilization procedure*, that consist in using a specialized read operation to obtain all the versions in the causal past from the tails of the corresponding chains (this may involve waiting until such versions are stable).

As soon as the dependencies have stabilized, the proxy uses consistent hashing to discover which data server is the head node of the chain associated with the target *key*, and forwards the PUT request to that node. The head then processes the PUT operation, assigning a new version to the object, and forwarding the request down the chain, as in the original chain replication protocol, until the k element of the chain is reached (we call this the *eager propagation phase*). At this point, a result is returned to the proxy, which includes the most recent version of the object and a *chainIndex* representing the k^{th} node. The proxy, in turn, forwards the reply to the client library. Finally, the library extracts the metadata and updates the corresponding entry in the table (updating the values of the object version and *chainIndex*).

In parallel with the processing of the reply, the update continues to be propagated in a lazy fashion until it reaches the tail of the chain. As we have noted, a data server may be required to process and forward write requests for different items. Updates being propagated in lazy mode have lower priority than operations that are being propagated in eager mode. This ensures that the latency of write operations of a given data item is not negatively affected by the additional replication degree of another item. When the PUT reaches the tail, the version written is said to be stable and an acknowledgment message is sent upwards in the chain (up to the head) to notify the remaining nodes. This message includes the key and version of the object so that a node can set that version of the object to a stable state.

F. Processing of Get Operations

Upon receiving the GET request, the client library consults the metadata entry for the requested *key* and forwards the request along with the *version* and the *chainIndex* to the client proxy. The client proxy uses the *chainIndex* included in the metadata to decide to which data server the GET operation is forwarded to. If *chainIndex* is equal to R, the size of the chain (*i.e.*, the version is stable), the request can be sent to any node in the chain at random. Otherwise, it selects a target data server t at random with an index from 0 (the head of the chain) to *chainIndex*. This strategy allows to distribute the load of the read requests among the multiple causally consistent servers. The selected server t processes the request and returns to the proxy the value of the data item, and the version read. Then the client proxy returns the value and the metadata to the client library which, in turn, uses this information to update its local metadata. Assume that a GET operation obtains version *newversion* from node with index *tindex*. The metadata is updated as follows: i) If the *newversion* is already stable, *chainIndex* is set to R; ii) If *newversion* is the same as *pversion*, *chainIndex* is set to *max*(*chainIndex*,*tindex*); iii) If *newversion* is greater than *pversion*, *chainIndex* is set to *tindex*.

G. Fault-Tolerance

The mechanisms employed by ChainReaction to recover from the failure of a node are the same as in chain replication. However, unlike the original chain replication, we can continue to serve clients even if the tail fails. If a node fails, two particular actions are taken: i) Chain recovery by adding to the tail of the chain a node that already is in the system (i.e., recover the original chain size); ii) Minimal chain repair for resuming normal operations (with a reduced number of nodes). Moreover, a node can later join the system (and the DHT) for load balance and distribution purposes.

In our system a chain with R nodes can sustain $R-k$ node failures, as it cannot process any PUT operation with less than k nodes. When a node fails a chain must be extended, therefore a node is added in the tail of the chain. To add this node, we must guarantee that the current tail (T) propagates its current state to the new tail (T^+). During the state transfer T^+ is in a quarantine mode and all new updates propagated by T are saved locally for future execution. When the state transfer ends, node T^+ is finally added to the chain and applies pending updates sent by T. Moreover, we can have the following 3 types of failures and corresponding repairs:

Head Failure: When the head node fails (H), its successor (H^+,) takes over as the new head, as H^+ contains most of the previous state of H. All updates that were in H but were not propagated to H^+ are retransmitted by the client proxy when the failure is detected.

Tail Failure: The failure of a tail node (T), its easily recovered by replacing the tail with T predecessor, say T^-. Due to the properties of the chain, T^- is guaranteed to have newer or equal state to the failing tail T.

Failure of a middle node: When a middle node fails (X) between nodes A and B, the failure is recovered by connecting A to B without any state transfer, however node A may have to retransmit some pending PUT operations that were sent to X but did not arrive to B. Since our solution only allows for version divergence to be one level deep, node A has to retransmit at most a single version per object (the same occurs during state transfers). The failure of node with index k (or of its predecessor) is treated in the same way.

In all cases, failures are almost transparent to the client, that it will only notice a small delay in receiving the response mostly, due to the time required for detecting the failure of a node. It is worth noticing that the above procedures are also applied if a node leaves the chain in an orderly fashion (for instance, due to maintenance).

Finally, the reconfiguration of a chain, after a node leaving/crash or when a node joins, may invalidate part of the metadata stored by the client library, namely the semantics of the *chainIndex*. However, since the last version read is also stored in the metadata, this scenario can be easily and safely detected. If the node serving GET request does not have a version equal or newer than the last seen by the client, the request will be routed upwards in the chain until it finds a node that contains the required version (usually its immediate predecessor).

IV. Supporting Geo-Replication

We now describe how ChainReaction addresses a scenario where data is replicated across multiple datacenters. We support Geo-replication by introducing a minimal set of changes with regard to the operation on a single site. However, metadata needs to be enriched to account for the fact that multiple replicas are maintained at different datacenters and that concurrent write operations may now be executed across multiple datacenters simultaneously. We start by describing the modifications to the metadata and then we describe the changes to the operation of the algorithms.

First, the version of a data item is no longer identified by a single version number but by a *version vector* (similarly to what happens in classical systems such as Lazy Replication [19]). Similarly, instead of keeping a single *chainIndex*, a *chainIndexVector* is maintained, that keeps an estimate of how far the current version has been propagated across chains in each datacenter.

We can now describe how the protocols for PUT and GET operations needs to be modified to address Geo-replication. For simplicity of exposition, we assume that datacenters are numbered from 0 to $D - 1$, where D is the number of datacenters, and that each datacenter number corresponds to the position of its entry in the version vector and *chainIndexVector*.

A. Processing of Put Operation

The initial steps of the PUT operation are similar to the steps of the single datacenter case. Let's assume that the operation takes place in datacenter i. The operation is received by a client proxy, the dependency stabilization procedure executed, and then the request is forwarded to the head of the corresponding chain. The operation is processed and the object is assigned with a new version, by incrementing the ith entry of the version vector. The updated is pushed down in the chain until it reaches node k. At this point a reply is returned to the proxy, that initializes the corresponding *chainIndexVector* as follows: all entries of the vector are set to 0 (*i.e.*, the conservative assumption that only the heads of the sibling chains in remote datacenters will become aware of the update) except for the ith entry that is set to k. This metadata is then returned to the client library. In parallel, the update continues to be propagated lazily down in the chain. When the update finally reaches the tail, an acknowledgment is sent upward (to stabilize the update) *and* to the tails of the sibling chains in remote datacenters (since all siblings execute this procedure, the global stability of the update is eventually detected in all datacenters).

Also, as soon as the update is processed by the head of the chain, the update is scheduled to be transferred in background to the remote datacenters. When a remote update arrives at a data center, it is sent to the head of the corresponding chain. If the update is more recent than the update locally known, it is propagated down the chain. Otherwise, it is discarded as it has already been superseded by a more recent update.

It is worth noting that each datacenter may configure a different value of k for the local chain, as the configuration of this parameter may depend on the characteristics of the hardware and software being used in each datacenter.

B. Processing of Get Operation

The processing of a GET operation in a Geo-replicated scenario is mostly identical to the processing in a single datacenter scenario. The only difference is that, when datacenter i receives a query, the set of potential targets to serve the query is defined using the ith position of the *chainIndexVector*. Finally, it may happen that the head of the local chain does not have yet the required version (because updates are propagated among different datacenters asynchronously). In this case, the GET operation can be redirected to another datacenter or blocked until a fresh enough update is available.

C. Conflict Resolution

Since the metadata carries dependency information, operations that are causally related with each other are always processed in the right order. In particular, a read that depends (even if transitively) from a given write, will be blocked until it can observe that, or a subsequent write, even if it is submitted to a different datacenter.

On the other hand, concurrent updates can be processed in parallel in different datacenters. However, similarly to many other systems that ensure convergence of conflicting object versions, ChainReaction's conflict resolution method is based on the last writer wins rule [20]. A conflict resolution mechanism is needed to ensure the *causal+* consistency of concurrent write operations. For this purpose, each update is also timestamped with the physical clock value of the proxy that receives the request. Note that timestamps are only used as a tiebreak if operations are concurrent. Therefore, physical clocks do not need to be tightly synchronized although, for fairness, it is desirable that clocks are loosely synchronized (for instance, using NTP). Finally, if two concurrent operations happen to have the same timestamp, the identifier of the datacenter is used as the last tiebreaker.

D. Fault-Tolerance over the Wide-Area

In ChainReaction, we have opted to return from a PUT operation as soon as it has been propagated to k nodes in a single datacenter. Propagation of the values to other datacenters is processed asynchronously. Therefore, in the rare case a datacenter becomes unavailable before the updates are propagated, causally dependent requests may be blocked until the datacenter recovers. If the datacenter is unable to recover, those updates may then been lost.

There is nothing fundamental in our approach that prevents the enforcement of stronger guarantees. For instance, the reply could be postponed until an acknowledgment is received from d datacenters, instead of waiting just for the acknowledgment of the local k^{th} replica (the algorithm would need to be slightly modified, to trigger the propagation of an acknowledgment when the update reaches the k^{th} node in the chain, both for local and remote updates). This would ensure survivability of the update in case of disaster. Note that the client can always re-submit the request to another datacenter if no reply is received after some preconfigured period of time. Although such extensions would be trivial to implement, they would impose an excessive latency on PUT operation, so we have not implement them. In a production environment, it could make sense to have this as an optional feature, for critical PUT operations.

V. Providing Get-transactions

The work presented in [8] has introduced a transactional construct which enables a client to read multiple objects in a single operation in a *causal+* consistent manner. This construct is named GET-TRANSACTION, an operation which can significantly simplify the work of developers, as it offers a stronger form of consistency on read operations over multiple objects. To use this transactional operation a client must issue a call to the Client Library through the following interface:

val1, ..., valN $\leftarrow$ GET-TRANSACTION (key1, ..., keyN)

Consider a scenario where client c_1 updates two objects. More precisely, c_1 updates twice objects X and Y, exactly in this order, creating the following sequence of causally related updates $x_1 \rightarrow y_1 \rightarrow x_2 \rightarrow y_2$. Lets now assume that another client c_2 concurrently reads objects X and Y, also in this sequence. If c_2 reads values x_1 and y_2 this does not violate causality. However, each of these values belongs to a different snapshot of the database, and this may violate the purposes of client c1. For interesting examples of the potential negative effects of reads from different snapshots, we refer the reader to [8].

To support GET-TRANSACTION operations the system must ensure that no concurrent writes are processed by the head of any of the chains that maintain one of the requested objects, while the GET-TRANSACTION is being processed. For this purpose, our implementation uses a sequencer process similar to the one used in [21]. All PUT operations and reads that are part of a certain GET-TRANSACTION are attributed a sequence number and are enqueued in each chain head queue considering the sequence numbers of operations. An interesting feature of this scheme is that, in opposition to [8], we do not require the existence of 2 rounds to process a GET-TRANSACTION.

With this in mind, a GET-TRANSACTION is processed as follows. The client proxy receives the GET-TRANSACTION and requests a sequence number for each chain where the keys are stores. Then, the individual reads are sent to the head of the corresponding chains the value is returned to the proxy. The proxy waits for all values, along with the corresponding metadata, assembles a reply, and sends it back to the client library. Similar to what happens with PUT and GET operations, upon receiving the reply from the client proxy, the client library extracts the metadata enclosed in the reply, and updates its local metadata. Notice that the metadata enclosed in the reply is equivalent to the metadata of multiple GET operations.

GET-TRANSACTIONS in a Geo-replicated scenario have the following additional complexity. Assume that a GET-TRANSACTION is being processed by datacenter i but it includes dependencies from values read in a different datacenter. Such updates may have not yet been propagated to datacenter i when the GET-TRANSACTION is processed. In this case, the read is aborted and retried in a (slower) two-phase procedure. First, the proxy reads all dependencies that have failed from the corresponding heads, using a specialized (blocking) read operation (the same as used in the *dependency stabilization procedure*) but directed to the head of the chains. Then, the GET-TRANSACTION is reissued as described above (and it is guaranteed to succeed).

VI. Implementation Issues

We have implemented ChainReaction on top of a version of FAWN-KV, that we have optimized. These optimizations were mostly related to the client proxies (frontends), key distribution over the chains, and read/write processing improving the overall performance of the FAWN-KV system. We also extended FAWN-KV to support multi-versioned objects in order to implement some aspects of our solution.

To efficiently encode and transmit dependency information between datacenters, we resort to an implementation of Adaptable Bloom Filters [22]. To this end, whenever a client issues a *get* or *put* operation, ChainReaction returns to that client, as part of the metadata, a bloom filter which encodes the identifier of the accessed object. This bloom filter is stored by the Client Library in a list named *AccessedObjects*. When the client issues a *put* operation, it tags its request with a bloom filter, named *dependency filter*, which is locally computed by the client library by performing a *binary* OR over all bloom filters locally stored in the *AccessedObjects* set. Upon receiving the reply, the ClientLibrary removes all bloom filters from the local *AccessedObjsects* set, and stores the bloom filter encoded in the returned metadata.

The *dependency filter* tagged by the Client Library on the *put* request, and the bloom filter that is returned to the issuer of the PUT (we will refer to this bloom filter as *reply filter* in the following text), are used by the datacenter that receives the PUT operation as follows: When a PUT request is propagated across datacenters it is tagged with both the *dependency filter* and the *reply filter* that are associated with the local corresponding PUT request. On the remote datacenter the client proxy receives the PUT request and places it in a waiting queue for being processed in the near future.

The two bloom filters associated with PUT requests encode causal dependencies among them. If a *wide-area-put* request op_1 has a *dependency filter* that contains all bits of a *reply filter* associated with another *wide-area-put* request op_2, we say that op_2 is potentially causally dependent of op_1. We say *potentially* because bloom filters can provide false positives, as the relevant bits of the *dependency filter* of op_1 can be set to one due to the inclusion of other identifiers in the bloom filter. The use of adaptable bloom filters allows us to trade the expected false positive rate with the size of the bloom filters. In our experiments, we have configured the false positive rate of bloom filters to 10%, which resulted in bloom filters with 163 bits.

VII. Experimental Evaluation

In this section we present experimental results, including comparative performance measures with three other systems: FAWN-KV, Cassandra, and COPS. We conducted experiments in four distinct scenarios, as follows: i) we have first assessed the throughput and latency of operations on ChainReaction in a single datacenter, and compare its results with those of FAWN-KV and Cassandra; ii) then we have assessed the performance of the system in a

Geo-replicated scenario (using 2 virtual datacenters), again comparing the performance with FAWN-KV and Cassandra; iii) we measured the performance of ChainReaction using a custom workload able to exercise GET-TRANSACTIONS; iv) finally, we measured the size of the metadata required by our solution and the overhead that it incurs. Throughput results were obtained from five independent runs of each test. Latency results reflect the values provided by YCSB in a single random run. Finally, the results from the metadata overhead e were obtained from ten different clients. Confidence intervals are plotted in all figures. The interested reader can refer to the Thesis where additional experimental results are presented and discussed, including operation latency results and metadata overhead results (not presented here due to the lack of space).

A. Single Datacenter Scenario

We first compare the performance of ChainReaction against FAWN-KV [3] and Apache Cassandra 0.8.10 in a single datacenter scenario. For sake of fairness, in the comparisons we have used the version of FAWN-KV with the same optimizations that we have implemented for ChainReaction. Our experimental setup uses 9 data nodes plus one additional independent node to generate the workload. Each node runs Ubuntu 10.04.3 LTS and has 2x4 core Intel Xeon E5506 CPUs, 16GB RAM, and 1TB Hard Drive. All nodes are connected by a 1Gbit Ethernet network. In our tests we used 5 different system configurations, as described below:

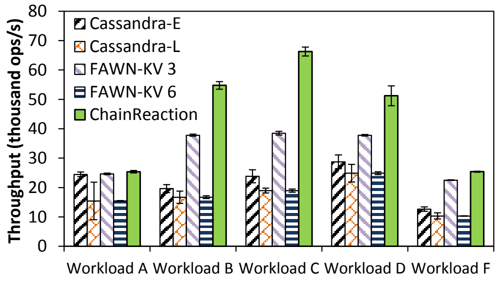
Cassandra-E and Cassandra-L: Deployments of Apache Cassandra configured to provide eventual consistency with a replication factor of 6 nodes. In the first deployment write operations are applied on 3 nodes while read operations are processed at a single node. In the second deployment both operations are processed by a majority of replicas (4 nodes).

FAWN-KV 3 and FAWN-KV 6: Deployments of the optimized version of FAWN-KV configured with a replication factor of 3 and 6 nodes, respectively, which provides linearizability (chain replication).

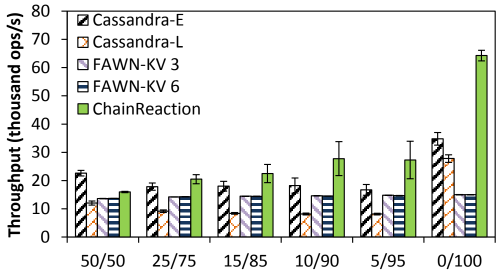
ChainReaction: Single Site deployment of ChainReaction, configured with $R = 6$ and $k = 3$. Provides *causal+* consistency.

All configurations have been subject to the Yahoo! Cloud Serving Benchmark (YCSB) version 0.1.3[23]. We choose to run standard YCSB workloads with a total of 1,000,000 objects. In all our experiments each object had a size of 1 Kbyte. We have also created a micro benchmark by using custom workloads with a single object varying the write/read ratio from 50/50 to 0/100. The latter allows assessing the behavior of our solution when a single chain is active. All the workloads were generated by a single node simulating 200 clients that, together, submit a total of 2,000,000 operations.

The throughput results are presented in Figure 2. Figure 2(a) shows that ChainReaction in a single datacenter outperforms both FAWN-KV and Cassandra in all standard YCSB workloads. In workloads A and F (which are write-



(a) Standard YCSB Workloads.



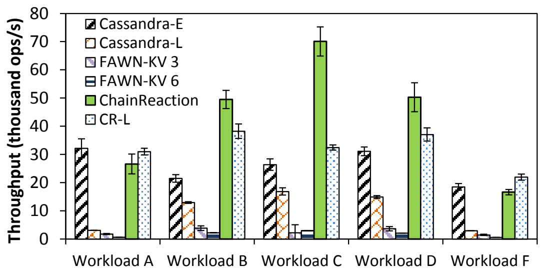
(b) Custom Workloads (single object).

Figure 2. Throughput (single site).

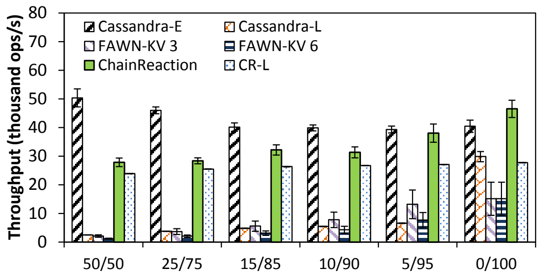
intensive) the performance of ChainReaction approaches that of Cassandra-E and FAWN-KV 3. This is expected, since ChainReaction is not optimized for write operations. In fact, for write-intensive workloads, it is expected that our solution under-performs when compared to FAWN-KV, given that ChainReaction needs to write on 6 nodes instead of 3 and also has to make sure, at each write operation, that all dependencies are stable before executing the next write operation. Fortunately, this effect is compensated by the gains in the read operations.

For workloads B and D, which are read-heavy, one expects ChainReaction to outperform all other solutions. Indeed, the throughput of ChainReaction in workload B is 178% better than that of Cassandra-E and 45% better than that of FAWN-KV 3. Performance results for workload D (Figure 2(a)) are similar to those of workload B. Notice that the latency of read operations for our solution is much better when compared with the remaining solutions (Figures **??** and **??**). Additionally, in workload C (read-only) ChainReaction exhibits a boost in performance of 177% in relation to Cassandra-E and of 72% in relation to FAWN-KV 3.

The micro benchmark that relies on the custom single object workloads has the purpose of showing that our solution makes a better use of the available resources in a chain, when compared with the remaining tested solutions. In the write-heavy workload (50/50) one can observe that Cassandra-E outperforms our solution by 70%. This can be explained by the fact that Cassandra is highly optimized for write operations specially on a single object. However, when we rise the number of read operations our solution starts to outperform Cassandra by 13%, 20%, 34%, and 39% in workloads 25/75, 15/85, 10/90, and 5/95, respectively.



(a) Standard YCSB Workloads.



(b) Custom Workloads (single object).

Figure 3. Throughput (multiple sites).

Additionally, ChainReaction outperforms FAWN-KV 3 and FAWN-KV 6 in all single object custom workloads. The performance increases as the percentage of read operations grows. Moreover, the throughput of the latter systems is always the same, which can be explained by the fact that the performance is bounded by a bottleneck on the tail node. If a linear speedup was achievable, our solution operating with 6 replicas would exhibit a throughput 6 times higher than FAWN-KV on a read-only workload (0/100 workload) with a single object. Although the speedup is sub-linear. As depicted in Figure 2(b), the throughput is still 4.3 times higher than that of FAWN-KV 3. The sub-linear growth is due to processing delays in proxies and network latency variations.

B. Geo-Replication

To evaluate the performance of our solution in a Geo-replicated scenario, we ran the same systems, by configuring nodes in our test setup to be divided in two groups with high latency between them to emulate 2 distant datacenters. In this test setup, each datacenter was attributed 4 machines, and we used two machines to run the Yahoo! benchmark (each YCSB client issues requests to one datacenter). The additional latency between nodes associated to different datacenters, was achieved by introducing a delay of 120 ms (in RTT) with a jitter of 10 ms. We selected these values as we measured them with the PING command to www.facebook.com (Oregon) from our lab. Each system considered the following configurations:

Cassandra-E and Cassandra-L: Eventual-consistency with 4 replicas at each datacenter. In the first deployment write operations are applied on 2 nodes and read operations are processed at a single node. In the second deployment operations are processed by a majority of replicas at each datacenter (3 nodes in each datacenter).

FAWN-KV 3 and FAWN-KV 6: Deployment of FAWN-KV configured with a replication factor of 4 and 6, respectively. In this case the chain can have nodes in both datacenters.

ChainReaction: Deployment of our solution with a replication factor of 4 for each datacenter and a k equal to 2.

CR-L: We introduced a new system deployment that consists of ChainReaction configured to offer linearizability on the local datacenter with a replication factor of 4 nodes. This deployment allows to compare the performance with systems that offer stronger local guarantees and weaker guarantees over the wide-area (in particular, COPS).

We employed the same workloads as in the previous experiments. However, in this case we run two YCSB clients (one for each datacenter) with 100 threads each. We also divided the workload among the two sites, meaning that each workload generator performs 1,000,000 operations on top of 1,000,000 objects. We aggregated the results of the two clients and present them in the following plots.

The throughput results are presented in Figure 3. Considering the standard YCSB workloads, we can see that ChainReaction outperforms the remaining solutions in all workloads except the write-heavy workloads (A and F) where Cassandra-E and CR-L are better. These results indicate that ChainReaction, Cassandra-E, and CR-L are the most adequate solutions for a geo-replicated deployment. The difference in performance between our solution and Cassandra-E is due to the fact that Cassandra offers weaker guarantees that our system and is also optimized for write operations resulting in an increase in performance. When comparing with CR-L our system needs to guarantee that a version is committed before proceeding with a write operation while CR-L does not, leading to some delay in write operations.

On read-heavy workloads (B and D), our solution surpasses both Cassandra-E and CR-L achieving 56%/22% better throughput in workload B and 38%/26% better performance in workload D. Finally, on workload C our solution exhibits an increase in performance of 62% and 53% in comparison with Cassandra-E and CR-L, respectively.

The low throughput of Cassandra-L and both FAWN-KV deployments is due to the fact that write operations always have to cross the wide-area network, inducing a great latency in operations. Moreover, in FAWN-KV (original chain replication) when the objects' chain tail is on a remote datacenter, read operations on that objects must cross the wide-area. Additionally, ChainReaction has a significantly higher throughput than FAWN-KV 3 ranging from 1,028% (workload F) to 3,012% (workload C) better (3 orders of magnitude). The comparison of the results for the remaining systems is similar.

The results for the micro benchmark (Figure 3(b)) in the Geo-replicated scenario are interesting because they show that the original Chain Replication protocol is not adaptable to a Geo-replicated scenario. The large error bars for both FAWN-KV deployments are a result of the difference in throughput in each datacenter. The client that has the tail of the object in the local datacenter has a better read throughput than the client on the remote datacenter, resulting in a great difference in each datacenter performance. Our solution outperforms FAWN-KV 3 in all workloads with a difference that ranges from 188% (Workload 5/95) to 1,249% (Workload 50/50).

Also, the results show that Cassandra-E outperforms our solution in all single object workloads with exception of the read-only workload (where our solution is 15% better). This happens because Cassandra behaves better with a single object and is optimized for write operations.

C. *Support for Get-Transactions*

In this experiment we evaluate the performance of GET-TRANSACTION operations in the Geo-replicated scenario. In this case we only executed ChainReaction (the other solutions do not support this operation) deployed in the 8 machines (4 in each simulated datacenter) like in the previous scenario. We have attempted to perform similar tests with COPS unfortunately, we were unable to successfully deploy this system across multiple nodes. We have created three custom workloads and changed the YCSB source in order to issue GET-TRANSACTION operations. The created workloads comprise the following distribution of write, read and GET-TRANSACTION operations: 10% writes, 85% reads, 5% GET-TRANSACTIONS on workload 10/85/5, 5% writes, 90% reads, 5% GET-TRANSACTIONS on workload 5/90/5, and 95% reads, 5% GET-TRANSACTIONS on workload 0/95/5. A total of 500,000 operations were executed over 10,000 objects, where a GET-TRANSACTION includes 2 to 5 keys (chosen randomly). This workload was executed by 2 YCSB clients (one at each datacenter) with 100 threads each.

Results depicted on Figure 4 show that the throughput for executed workloads is quite reasonable. We achieve an aggregate throughput that approximates of 12,000 operations per second in all workloads showing that the percentage of write and read operations do not affect the performance of GET-TRANSACTIONS and vice-versa.

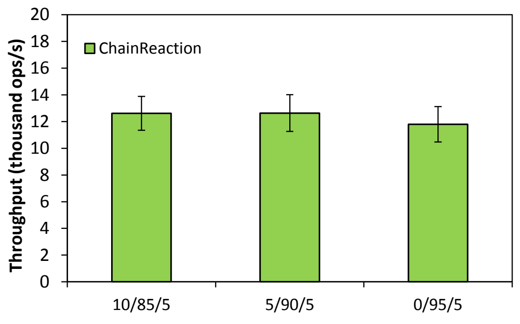
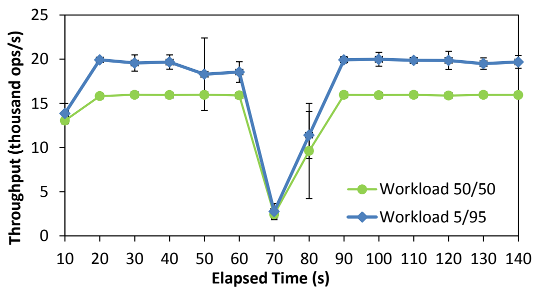


Figure 4. Throughput for GET-TRANSACTION workloads.

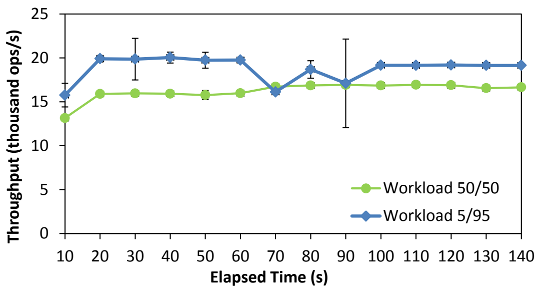
In terms of operation latency we can state (see full Thesis for more details) that the introduction of GET-TRANSACTIONS does not affects the latency of write and read operations in the three workloads. On the other hand, since we give priority to the other two operations, the average latency for GET-TRANSACTIONS is in the order of approximately 400 ms (which we consider acceptable from a practical point of view).

D. *Fault-Tolerance Experiments*

To assess the behavior of our solution when failures occur we deployed ChainReaction in a single data center with 9 data nodes and a single chain, with a replication factor of 6 and a k equal to 3. A single chain was used so that the failures could be targeted to the different zones of the chain. We used the custom-made workloads 50/50 and 5/95 to measure the average throughput of our solution during a period of 140 seconds. During the workload we failed a single node at 60 seconds. We tested two scenarios of failure: a) a random node between the head and node k (including k); b) a random node between k and the tail (excluding k). The workloads were executed with 100 client threads that issue 3,000,000 operations over a single object.



(a) Node between the head of the chain and node k.



(b) Node between $k$and the tail of the chain.

Figure 5. Throughput in face of failures.

The results for the average throughput during execution time can be observed in Figure 5. In the first scenario, depicted by Figure 5(a), one can observe that the failure of a node between the head of the chain and node k results in a drop in throughput. This drop reaches approximately 2000 operations per second in both workloads and is due to the fact that write operations are stalled until the failure is detected and the chain is repaired. Also, 20 seconds after the failure of the node the throughput starts increasing reaching its initial peak 30 seconds after the failure. The results for the second scenario, depicted in Figure 5(b), show that the failure of a node after node k has a reduced impact in the performance of the system, as the write operations can terminate with no problems. Also, the variations of the throughput during the repair of the chain are due to the fact that read operations are processed by only 5 nodes while the chain is being repaired.

VIII. Conclusions

The work described in the thesis proposes ChainReaction, a distributed key-value store that offers high-performance, scalability, and high-availability. Our solution offers the recently formalized *causal+* consistency guarantees which are useful for programmers. Similarly to COPS, we also provide a transactional construct called GET-TRANSACTION, that allows to get a consistent view over a set of objects. This datastore can be deployed in a single datacenter scenario or across multiple datacenters, in a Geo-replicated scenario. We have implemented a prototype of ChainReaction and used the Yahoo! Cloud Serving Benchmark to test our solution against existing datastores. Experimental results using this testbed show that ChainReaction outperforms Cassandra and FAWN-KV in most workloads that were run on YCSB.

Acknowledgments

I am thankful to the members of the Distributed Systems Group at INESC-ID who collaborated in building and improving this thesis, namely, João Leitão, João Paiva, Oksana Denysyuk, Maria Couceiro and Miguel Branco. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/ EEI/ LA0021/ 2011, and via the project HPCI (PTDC/ EIA-EIA/ 102212/ 2008).

References

[1] E. Brewer, "Towards robust distributed systems (abstract)," in *ACM PODC*, 2000, p. 7.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS OSR*, vol. 44, pp. 35–40, April 2010.

[3] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: a fast array of wimpy nodes," *Comm. ACM*, vol. 54, no. 7, pp. 101–109, Jul. 2011. [Online]. Available: http://doi.acm.org/10.1145/1965724.1965747

[4] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SOSP*, 2007, pp. 205–220.

[5] B. 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," in *VLDB*. VLDB Endowment, August 2008, pp. 1277–1288.

[6] J. Terrace and M. J. Freedman, "Object storage on craq: high-throughput chain replication for read-mostly workloads," in *Proc. USENIX*, Berkeley, USA, 2009.

[7] Y. Sovran, R. Power, M. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *ACM SOSP*, 2011, pp. 385–400.

[8] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *ACM SOSP*, 2011, pp. 401–416.

[9] R. van Renesse and F. Schneider, "Chain replication for supporting high throughput and availability," in *USENIX OSDI*, Berkeley, USA, 2004.

[10] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, pp. 463–492, July 1990.

[11] L. Lamport, "The part-time parliament," *ACM TOCS*, vol. 16, pp. 133–169, May 1998.

[12] B. W. Robert Escriva and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store for cloud computing," CSD, Cornell University, Tech. Rep., December 2011.

[13] G. Laden, R. Melamed, and Y. Vigfusson, "Adaptive and dynamic funnel replication in clouds," in *ACM LADIS*, September 2011.

[14] J. Baker, C. Bond, J. Corbett, 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*, 2011, pp. 223–234.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. of the ACM*, vol. 21, pp. 558–565, July 1978.

[16] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *ACM SOSP*, 1997, pp. 288–301.

[17] N. Belarami, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "PRACTI replication," in *USENIX NSDI*, May 2006, pp. 59–72.

[18] C. Lesniewski-Laas, "A sybil-proof one-hop DHT," in *ACM SNS*, 2008, pp. 19–24.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM TOCS*, vol. 10, pp. 360–391, November 1992.

[20] R. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM TODS*, June 1979.

[21] D. Malkhi, M. Balakrishnan, J. Davis, V. Prabhakaran, and T. Wobber, "From paxos to corfu: a flash-speed shared log," *SIGOPS OSR*, vol. 46, no. 1, pp. 47–51, Feb. 2012.

[22] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *IEEE PRDC*, 2009, pp. 307–313.

[23] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM SOCC*, 2010, pp. 143–154.