ChainReaction: a Causal+ Consistent Datastore based on Chain Replication

Sérgio Almeida
INESC-ID, Instituto Superior Técnico, U. Técnica de Lisboa
sergiogarrau@gsd.inesc-id.pt

João Leitão
CITI / DI-FCT-Universidade Nova de Lisboa *and* INESC-ID, IST, UTL
jc.leitao@fct.unl.pt

Luís Rodrigues
INESC-ID, Instituto Superior Técnico, U. Técnica de Lisboa
ler@ist.utl.pt

Abstract

This paper proposes a Geo-distributed key-value datastore, named ChainReaction, that offers *causal+* consistency, with high performance, fault-tolerance, and scalability. ChainReaction enforces causal+ consistency which is stronger than eventual consistency by leveraging on a new variant of chain replication. We have experimentally evaluated the benefits of our approach by running the Yahoo! Cloud Serving Benchmark. Experimental results show that ChainReaction has better performance in read intensive workloads while offering competitive performance for other workloads. Also we show that our solution requires less metadata when compared with previous work.

Categories and Subject Descriptors C.2.4 [*Computer Systems Organization*]: Distributed Systems; C.4 [*Performance of Systems*]

Keywords Key-value storage, causal+ consistency, Geo-replication, chain-replication

1. Introduction

The trade-offs among consistency and performance, in particular for systems supporting Geo-replication, introduce some of the most challenging aspects in the design of datastores for cloud-computing applications. A subset of these trade-offs have been captured by the well-known CAP Theorem [5], which states that it 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, 6, 10, 14, 17, 22, 23]. Some solutions opt to weaken consistency, in order to achieve the desired efficiency. Unfortunately, weak 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.

In this context, chain-replication [25], is a very interesting mechanism that is able to provide linearizability, while enabling a high degree of parallelization in the processing of write operations issued by different clients. The inherent simplicity and parallelism of this approach has motivated us to further study mechanisms that can enable chain-replication to offer high performance and strong consistency guarantees in Geo-replicated scenarios.

As a result of this, we propose a novel datastore design, named ChainReaction. Our solution relies on a novel variant of chain-replication that offers the *causal+* consistency criteria (recently formalized in [17, 18]) and is able to leverage 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. Additionally, our variant of chain replication enables us to more efficiently deal with metadata information that encodes causal dependencies between operations, when compared with a state of the art solution that also offers causal+ consistency. For this purpose, we implement a stabilization procedure that allows to preserve causal guarantees while keeping the metadata overhead low (both at the client and at the datastore levels). Furthermore, ChainReaction can be deployed either on a single datacenter or on Geo-replicated scenarios, over multiple datacenters. Finally, and similarly to [17], 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 running the Yahoo! Cloud Serving Benchmark to a prototype deployment that includes ChainReaction, Apache Cassandra [14], FAWN-KV [2], and a system that emulates COPS [17].

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

The rest of the paper is organized as follows. Section 2 addresses related work. Section 3 discusses the operation of ChainReaction in a single datacenter, Section 4 presents our extensions to support Geo-replication, and Section 5 describes the support for GET-TRANSACTIONS. Section 6 provides a brief discussion on the current implementation. Section 7 presents the results of the experimental evaluation. Section 8 concludes the paper.

2. Related Work

A datastore for Geo-replicated systems must address the occurrence of faults, client locality (latency), and avoid blocking in face of network partitions. Therefore, there is an inherent necessity for some form of replication, including replication over the wide-area. Ideally, such a datastore would provide linearizability [11], 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 usually sacrifice strong consistency in order to provide both availability, and efficiency.

Relevant examples include Yahoo's PNUTS [6] which is a Geo-replicated key-value store that offers storage for large-scale applications. This system tries to provide guarantees that lie between strong and eventual consistency by implementing the per-record timeline consistency model [6]. PNUTS provides an API containing a new set of operations that allows to achieve different degrees of consistency. This however somewhat limits the ability of existing applications to operate over PNUTS. Recent work [22], introduced a new Geo-distributed key-value datastore, named Walter, that supports transactions. This datastore implements a new transactional consistency model called parallel snapshot isolation. In order to enforce this consistency model, Walter resorts to the use of counting sets and also of per-object preferred sites that must handle all write operations over that objects. In sharp contrast, ChainReaction allows write operations to be directed to any site, which is essential to provide service to users in the face of the unavailability of some datacenters (for instance because of network partitions, or a catastrophic event).

Chain replication [25] is a data replication technique that provides linearizability, 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 [2] and Hyperdex [9] are two datastores that offer strong consistency using chain-replication as the main replication technique. CRAQ [23] is also based on chain replication but, for performance, supports eventual consistency by not constraining reads to be executed on the tail.

Apache Cassandra [14] uses a quorum technique to maintain replicas consistent. These quorums can be configured to provide different forms of consistency. Quorums are also used by Amazon's Dynamo [10], that provides eventual consistency over the wide-area and resorts to conflict resolution mechanisms based on vector clocks. Also the work described in [12] 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 [3] 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 [17] is a datastore designed to provide high scalability over the wide-area. For this purpose, COPS has proposed a weak consistency model that, contrary 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* that allow a client to read several mutually consistent objects in single operation [17].

3. 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 4. 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.

3.1 Consistency Model

We have opted to offer the *causal+* consistency model [4, 17, 20]. We have selected *causal+* because it provides a good trade-off between consistency and performance. Contrary to linearizability, *causal+* allows for a reasonable amount of parallelism in the processing of concurrent requests while

[1] In fact, a similar consistency model has been used before, for instance in [4, 18, 20] , but was only coined as *causal+* in [17].

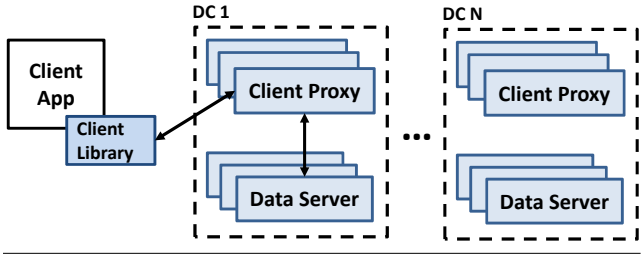


Figure 1. Overview of the ChainReaction architecture.

still respecting the causal dependencies associated with these request. Although replicas can temporarily diverge due to concurrent updates at different sites, they are guaranteed to eventually converge (a guarantee that is not provided by 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 [17], our system also supports GET-TRANSACTIONS, that allows an application to obtain a *causal+* consistent snapshot of a set of objects.

3.2 Architecture Overview

The architecture of ChainReaction is loosely based on the architecture of the FAWN-KV system [2] which relies on (classical) chain replication. 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's 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 [16]. 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.

3.3 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 more efficiently use 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 [25]. 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* is propagated to the tail of the chain, 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, which minimizes the interference, due to the larger replication set for a given object, over the propagation of write request in other chains.

We note that our use of the parameter k may, at first sight, appear equivalent to the use of a write quorum (as in quorum based systems). However, in contrast with a quorum system, the k nodes that are required to process the data are not arbitrary but instead defined by the chain topology. Additionally, replicas that are allowed to process a particular read are chosen considering their position in the chain.

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 their last read requests for each object they have read.[2] 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 ([17] 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 been fully propagated over their respective chains 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.

3.4 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** ← **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 object that was accessed by that particular client. 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 may update the metadata as a result of executing such an operation.

3.5 Processing of Put Operations

We now provide a detailed description on how PUT operations are executed in ChainReaction. To simplify the exposition we rely on the following definition:

DC-Write-Stable(d) A write operation is said to be DC-Write-Stable(d) for a datacenter d, when the write has been propagated and applied to all nodes in the chain located in datacenter d that is responsible for the object targeted by the write operation. We note that when a write operation for the version of an object o become DC-Write-Stable(d), no client is able to read a previous version of object o in datacenter d.

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 relates to the GET operations performed over any objects since that PUT. Metadata is only maintained for objects whose version write operation is not yet known to be DC-Write-Stable(d); DC-Write-Stable(d) versions do not put constraints on the execution of PUT or GET operations on datacenter d (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 causally depends has become DC-Write-Stable(d) (*i.e.*, the version has been applied to the respective tail on datacenter d). This ensures that no client is able to read mutually inconsistent versions of two distinct objects. To ensure this, when a client proxy receives a PUT request, it has to ensure that all write operations associated with the dependencies of that PUT have become DC-Write-Stable(d). This is achieved using a *dependency stabilization procedure*, that consists of issuing a blocking read operation for each object versions that is in the causal history of the client issuing that operation. Each of these read operations is directed to the tail of the appropriate chain and carries the version of that particular object from which the PUT depends, and only returns to the client proxy when that version, or a newer one, has reached that node.

As soon as all write operations associated with the dependencies have become DC-Write-Stable(d), 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, which includes the most recent version of the object and a *chainIndex* representing the k^{th} node is returned to the proxy. 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 chains. 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 becomes DC-Write-Stable(d), an acknowledgment message is sent

[2] Notice that if a client crashes and recovers it can be seen as a new client without any causal history.

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.

3.6 Processing of Get Operations

Upon receiving the GET request, the proxy consults the metadata entry for the requested *key* and forwards the request along with the *version* and the *chainIndex* to a data server. 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, the request can be sent to any node in the chain at random. Otherwise, the proxy 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 read requests among the multiple servers whose state is causally consistent. 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*.

3.7 Fault-Tolerance

The mechanisms employed by ChainReaction to recover from the failure of a node are the same as in the original 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 fewer than k nodes. When a node fails a chain must be extended, therefore a node is added to 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. We note that nodes located before T in the chain can still process and propagate PUT operations while the chain is being repaired. 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^-. Because of 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 (X) fails between nodes A and B, the chain is repaired 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. 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 only notices a small delay in receiving the response mostly because of 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, because of maintenance).

Finally, the reconfiguration of a chain, after a node leaves/crashes 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 detected. If the node serving a 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).

4. 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 write operations may now be executed across multiple datacenters concurrently. 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 [13]). Also, 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 also consider a new definition that captures the notion of DC-Write-Stable(d) across all datacenters:

Global-Write-Stable A write operation is said to be Global-Write-Stable when the write is DC-Write-Stable(d) for all datacenters d. We note that when a write operation for

the version of an object o become Global-Write-Stable, no client is able to read a previous version of object o.

We can now describe how the protocols for PUT and GET operations need 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 is the position of its entry in the version vector and *chainIndexVector*.

4.1 Processing of Put Operations

The initial steps of the PUT operation are similar to the steps of the single datacenter case. 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 a new version by incrementing the ith entry of the version vector. The update 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.*, which comes from a 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 notify elements of the chain that the operation has become DC-Write-Stable(d)) *and* to the tails of the sibling chains in remote datacenters (since all siblings tails execute this procedure, the update is eventually detected as being Global-Write-Stable 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 by forwarding the request to a local *remote-proxy*[3]. A *remote-proxy* combines several updates in a single *remote-update* that is then propagated to the *remote-proxies* located in other datacenters. To ensure that operations performed by clients in remote datacenters respect the causality between operations, one has to ensure that PUT operations are only applied - and therefore become visible - in remote datacenters when all versions of objects in their causal history have become DC-Write-Stable(d) in that particular datacenter d.

To achieve this we have to ensure that: i) causal dependencies of operations originally executed in other datacenters have become stable and ii) causal dependencies among operations in a given *remote-update* are respected. We now discuss how we ensure both these properties.

[3] For fault-tolerance, the remote-proxy should be replicated. We however do not address this issue explicitly as there are well known techniques that can be easily employed for this [21].

4.1.1 Dependencies among operations issued to different datacenters

To enforce dependencies among write operations that originate from different datacenters, we restrict the order in which remote-updates can be applied in each datacenter. Each *remote-proxy* maintains a version vector, named *remote proxy vector*, or simply *rvp*, that encodes the number of remote-updates that the *remote-proxy* has issued to other datacenters and the number of remote-updates originated from remote datacenters that itself has applied.

Considering this, when the *remote-proxy* in datacenter i issues a remote-update, it tags the remote-update with its local *rvp*, after which it increases the i^{th} position of the local *rvp*. When the *remote-proxy* of datacenter j receives this update, it compares its local *rvp* with the rvp_i enclosed in the update to verify that all positions with the exception of the j^{th} have the same (or higher) value. If this is true, the remote-update can be processed, otherwise the *remote-proxy* has to wait for the missing remote-updates. When a *remote-proxy* starts processing of a remote-update issued by datacenter i it sets the i^{th} position of its local *rvp* to the value of the i^{th} position of the rvp_i enclosed in the update. It is relevant to observe that this scheme allows for concurrent remote-updates to be processed in parallel on each datacenter.

We note that if two datacenters, say a and b, become unable to communicate with each other, while still being able to communicate with the remaining datacenters, a and b may become unable to apply remote-updates. To circumvent this problem, we enable *remote-proxies* to cache remote-updates from other datacenters so that they can retransmit them to *remote-proxies* that have missed them and that explicitly request them. Note however, that the *rvp*s enclosed in each remote-update encode enough information to enable each remote-proxy to locally garbage collect updates that have already been applied in every datacenter.

4.1.2 Dependencies among operations in a single remote-update

When a remote-update is processed by a *remote-proxy*, we must ensure that causal dependencies among individual put operations enclosed in that update are respected. A straightforward way to achieve this could be to apply individual PUT operation contained in a remote-update sequentially. This however would be highly inefficient, as no parallelization could be achieved. In order to attain a significant level of parallelism on the application of remote-updates, while at the same time enforcing causal dependencies, we resort to an efficient encoding technique that relies on Adaptable Bloom Filters [8]. We now discuss how this solution operates.

Whenever a client issues a *get* or *put* operation, ChainReaction returns to that client, as part of the metadata, a bloom filter that encodes the identifier (which is composed of the unique identifier and the version) 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 stored in its *AccessedObjects* set. Upon receiving the reply, the ClientLibrary removes all bloom filters from the local *AccessedObjects* 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 scheduled to be disseminated 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, when the *remote-proxy* processes a remote-update it puts all contained PUT in a waiting queue for being applied 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 on 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.

When a particular PUT is cleared for execution 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 is already superseded by a more recent update. This update is removed from the execution list by the *remote-proxy* when it is either discarded or it becomes DC-Write-Stable(d) in that datacenter.

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.

4.2 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 the required version (because updates are propagated among different datacenters asynchronously). In this case, the GET operation can either be redirected to another datacenter or blocked until a fresh enough update becomes locally available. Note that the execution of a GET offers the possibility for updating the chainIndex in the client library by taking into consideration the information that is locally available to the node that replied to the client concerning the Global-Write-Stable of the write operation associated with the version of the object being returned.

4.3 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 [24]. For this purpose, each update is timestamped when received by a proxy with a pair (c, s) where c is the clock of the proxy, and s its datacenter which allows to establish a total order among updates. Note that physical clocks do not need to be tightly synchronized although, for fairness, it is desirable that clocks be *loosely* synchronized (for instance, using NTP). If two updates have identical values for c, the value of s is used as a last tiebreaker. This approach offers a concurrency control that is more fair than simply relying on datacenters identifiers.

4.4 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 pre-configured 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.

5. Providing Get-transactions

The work presented in [17] has introduced a transactional construct, named GET-TRANSACTION, which enables a client to read multiple objects in a single operation in a *causal+* consistent manner. This construct 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} ← GET-TRANSACTION (key1, ..., keyN)

Consider a scenario where client c_1 makes multiple updates to two objects X and Y in the following causal order $x_1 \rightarrow y_1 \rightarrow x_2 \rightarrow y_2$. Assume that another client c_2 concurrently reads the same objects X and Y and observes the following values $x_1 \rightarrow y_2$. Although these reads do not violate causality, they 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 [17].

To support GET-TRANSACTION operations we must ensure that they are atomically ordered with respect to concurrent write operations. On the other hand, for efficiency, GET-TRANSACTION operations should not block write operations. We conciliate these goals as follows:

First, our implementation uses a sequencer process, similar to the one used in [19], that is local to each datacenter. This sequencer is used to order: i) all PUT operations and ii) reads that are part of a GET-TRANSACTION. The sequencer process maintains a different sequence number for each chain, and PUT operations are processed taking these numbers into account. Namely, to avoid delays, the most recent PUT operation is always processed by the head of the chain and late PUT operations are just used to create the old versions required to support GET-TRANSACTION.

Second, to avoid blocking of PUT operations during the execution of a GET-TRANSACTION, we resort to multiversion. This allows a proxy to read versions consistent to *causal+*, even if new versions are created concurrently. This is implemented by returning the version created by the PUT that preceded the GET-TRANSACTION, considering the sequence numbers of operations. An interesting feature of this scheme is that, in opposition to [17], we can avoid 2 rounds to process most 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 relevant keys are stored. These sequence numbers are assigned atomically by the sequencer. Then, the individual reads are sent to the head of the corresponding chains, which then return the value of the previous PUT. The proxy waits for all values, along with the corresponding metadata, assembles a reply, and sends it back to the client library. The metadata is processed by the client library in a manner similar to that described before for individual GET operations.

Due to the asynchrony of the system, it may happen that the PUT operation that was sequenced before the GET-TRANSACTION is not yet available at a particular chain head when the associated read request is processed. In this case, the processing of the operation is delayed, until the version becomes locally available or a timeout occurs. In the latter case, the chain head replies to the proxy that issued the request, aborting the GET-TRANSACTION. The proxy should then gather new sequence numbers from the sequencer and re-issue the GET-TRANSACTION to all head of the chains involved in the processing of that particular operation.

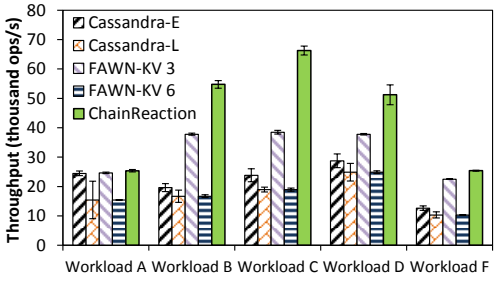
GET-TRANSACTIONS in a Geo-replicated scenario have the following additional complexity. Assume that a GET-TRANSACTION is being processed by datacenter i but it includes dependencies from values written or read in a different datacenter. Such updates may have not yet been propagated to datacenter i when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

6. Implementation Issues

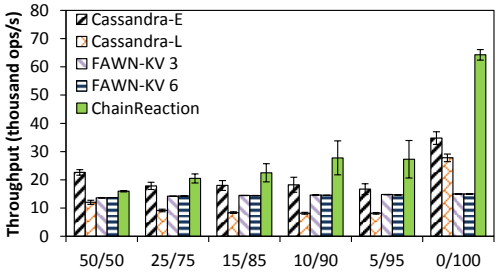
We have implemented ChainReaction on top of a version of FAWN-KV, that we have optimized. These optimizations were mostly related to the client proxies (frontends), key distribution over the chains, and read/write processing improving the overall performance of the FAWN-KV system. We also extended FAWN-KV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

7. 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 were obtained from ten different clients. Confidence inter-

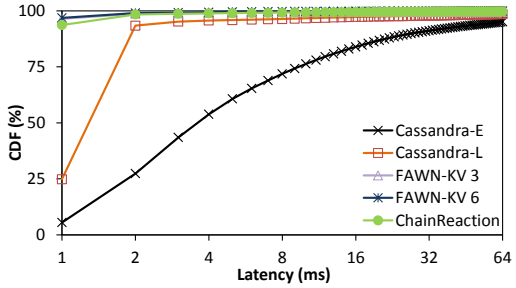


(a) Standard YCSB Workloads.

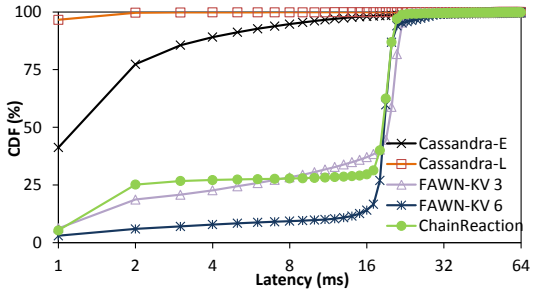


(b) Custom Workloads (single object).

Figure 2. Throughput (single site).



(a) Read Latency.



(b) Write Latency.

Figure 3. Latency CDF for Workload A (single site).

vals are plotted in all figures. The interested reader can refer to [1] where additional results are presented and discussed.

7.1 Single Datacenter Scenario

We first compare the performance of ChainReaction against FAWN-KV [2] 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).

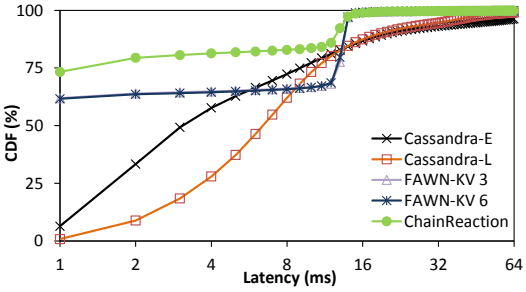
FAWN-KV 3 and FAWN-KV 6: Deployments of the optimized version of FAWN-KV configured with a replication factor of 3 and 6 nodes, respectively, which provides linearizability (chain replication).

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

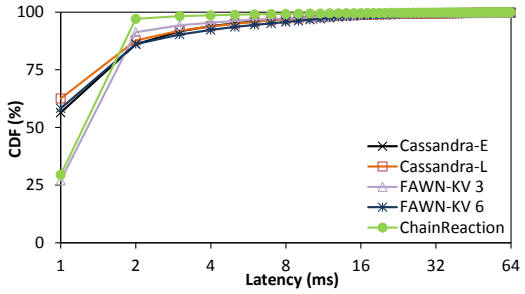
All configurations have been subject to the Yahoo! Cloud Serving Benchmark (YCSB) version 0.1.3[7]. We choose to run standard YCSB workloads with a total of 1,000,000 objects. In all our experiments each object had a size of 1 Kbyte. We have also created a micro benchmark by using custom workloads with 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. Latency Cumulative Distribution Function (CDF) results[4] for workloads A and B are presented in Figures 3 and 4. 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-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. This can be observed in the latency results for workload A in Figures 3(a) and 3(b). These figures also show that Cassandra exhibits a better write latency. Notice however, that Cassandra has much slower read operations than ChainReaction and FAWN-KV since it is optimized for write-heavy environments. In workload C, which

[4] Note that in all Latency CDF charts the CDF is in percentage and the latency is in a logarithmic scale.



(a) Read (Workload B).



(b) Write (Workload B).

Figure 4. Latency CDF for Workload B (single site).

is read only, FAWN-KV 6 exhibits a throughput that is approximately 50% of the throughput exhibited by FAWN-KV 3. This happens because in our implementation multiple objects may be stored in the same file. In FAWN-KV 6 each node has to manage twice the number of objects of each node in FAWN-KV 3, which increases the time required to scan files to read objects. Different implementations may not exhibit such difference.

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 4(a) and 4(b)). 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. In terms of latency one can see that ChainReaction always exhibits a better read latency than Cassandra-E having more operations to complete at lower latencies. We can also observe that as the number of reads increases the write latency of ChainReaction is also better than Cassandra-E write latency.

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. Latency results for single-object workloads are consistent with that of the standard YCSB workloads.

7.2 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 laboratory in Lisbon, Portugal. Each system considered the following configurations:

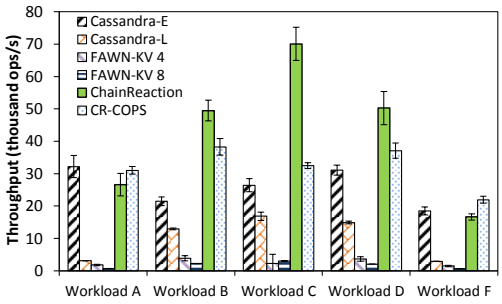
Cassandra-E and Cassandra-L: Eventual-consistency with 4 replicas at each datacenter. In the first deployment write operations are applied on 2 nodes and read operations are processed at a single node. In the second deployment operations are processed by a majority of replicas at each datacenter (3 nodes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

FAWN-KV 4 and FAWN-KV 8: Deployment of FAWN-KV configured with a replication factor of 4 and 8, respectively. In this case each chain is composed of nodes located in both datacenters.

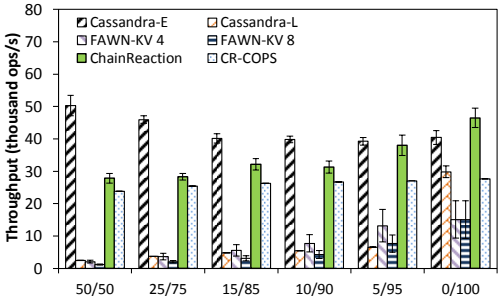
ChainReaction: Deployment of our solution with a replication factor of 4 for each datacenter and a k equal to 2.

CR-COPS: 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

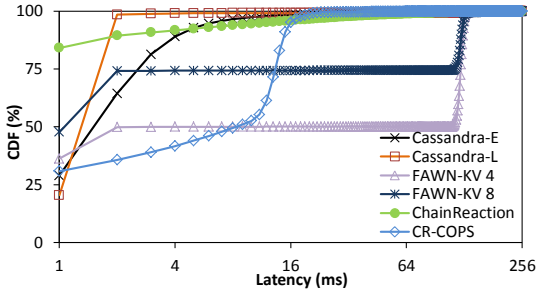


(a) Standard YCSB Workloads.

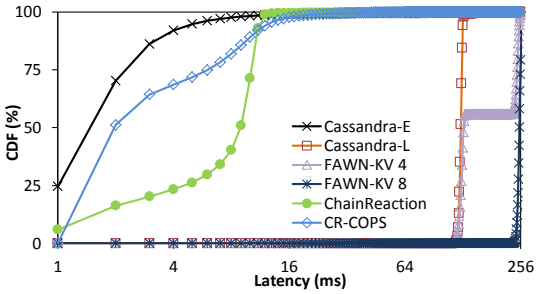


(b) Custom Workloads (single object).

Figure 5. Throughput (multiple sites).



(a) Read latency.



(b) Write latency.

Figure 6. Latency CDF for Workload A (multiple sites).

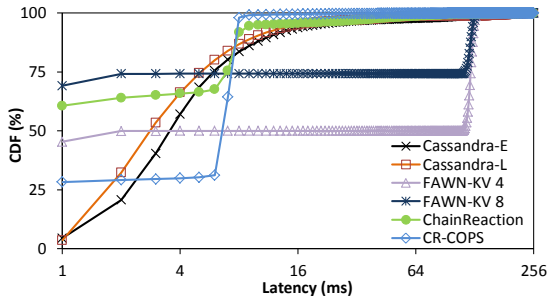
(one for each datacenter) with 100 threads each. We also divided the workload among the two sites, meaning that each workload generator performs 1,000,000 operations on top of 1,000,000 objects. We aggregated the results of the two clients and present them in the following plots.

The throughput results are presented in Figure 5. Latency Cumulative Distribution Function (CDF) results are presented in Figures 6 and 7. 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-COPS are better. These results indicate that ChainReaction, Cassandra-E, and CR-COPS 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-COPS our system needs to guarantee that a version is committed before proceeding with a write operation while CR-COPS does not, leading to some delay in write operations. In terms of latency in write-heavy workloads our system is slightly outperformed by Cassandra-E and CR-COPS.

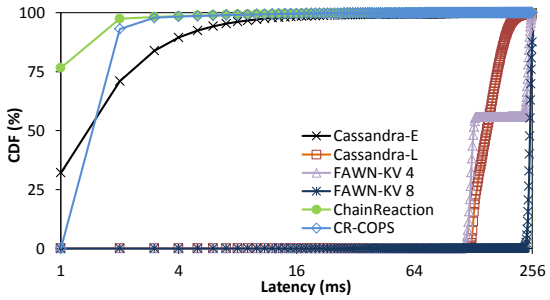
On read-heavy workloads (B and D), our solution surpasses both Cassandra-E and CR-COPS achieving 56%/22% better throughput in workload B and 38%/26% better performance in workload D. The latency results depicted in Figure 7 shows that our solution provides better write and read latency in the read heavy workload B than the other solutions. Finally, on workload C our solution exhibits an increase in performance of 62% and 53% in comparison with Cassandra-E and CR-COPS, respectively.

The low throughput of the Cassandra-L and FAWN-KV deployments is due to the fact that write operations always have to cross the wide-area network. Moreover, in FAWN-KV, when the 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 4 ranging from 1,028% (workload F) to 3,012% (workload C), i.e., up to 3 orders of magnitude better.

The results for the micro benchmark (Figure 5(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 4 in all workloads with a difference that ranges from 188% (Workload 5/95) to 1,249% (Workload 50/50). In terms of latency its possible to observe that 50% of write operations in FAWN-KV 3 take more than 100 ms to complete corresponding to the latency introduced between datacenters. The results for Cassandra-L and FAWN-KV 8 are similar. Finally, results show that Cassandra-E outperforms our solution in all single object workloads with exception of the read-only workload (where our solution is



(a) Read latency.



(b) Write latency.

Figure 7. Latency CDF for Workload B (multiple sites).

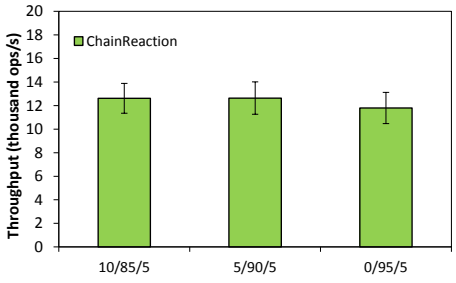
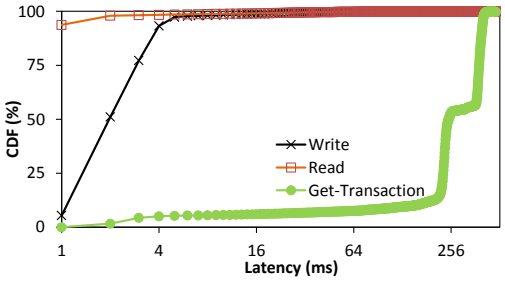


Figure 8. Throughput for GET-TRANSACTION.



(a) Workload 10/85/5.

Figure 9. Latency for GET-TRANSACTIONS.

15% better). This happens because Cassandra behaves better with a single object and is optimized for write operations.

7.3 Support for GetTransactions

In this experiment we evaluate the performance of GET-TRANSACTION operations in the Geo-replicated scenario. For this purpose we have executed ChainReaction (the other solutions do not support this operation) deployed in the 8 machines like in the previous scenario (4 in each simulated datacenter). 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.

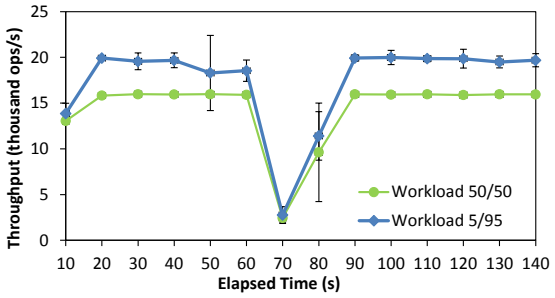
Results depicted on Figure 8 show that the throughput for executed workloads is quite reasonable. We achieve an aggregate throughput that approximates of 12,000 operations per second in all workloads showing that the percentage of write and read operations do not affect the performance of GET-TRANSACTIONS and vice-versa.

In terms of operation latency we can see on Figure 9 that the introduction of GET-TRANSACTIONS does not affects the latency of write and read operations in Workload 10/85/5 (results for the other workloads are similar). 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).

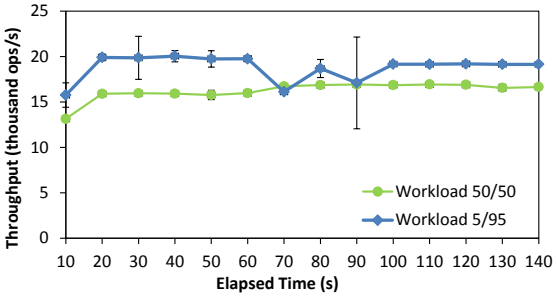
7.4 Fault-Tolerance Experiments

To assess the behavior of our solution when failures occur we deployed ChainReaction in a single datacenter 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.

The results for the average throughput during execution time can be observed in Figure 10. In the first scenario, depicted by Figure 10(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 de-

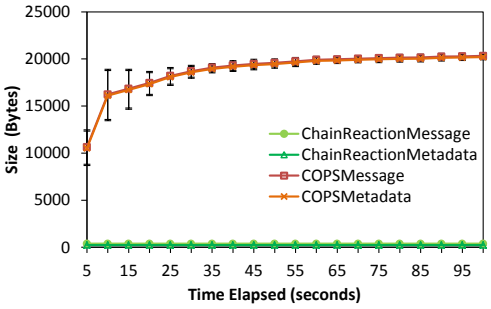


(a) Node between the head of the chain and node k.

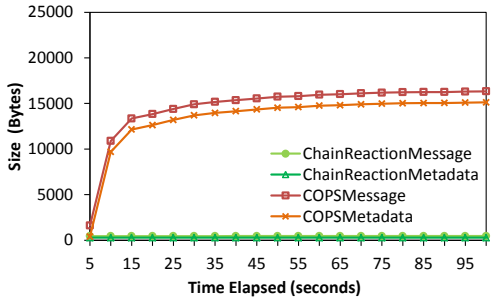


(b) Node between $k$and the tail of the chain.

Figure 10. Throughput in face of failures.



(a) Average size of one PUT message.



(b) Average size of one wide-area message.

Figure 11. Metadata overhead

tected 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 10(b), show that the failure of a node after node k has a low impact in the performance of the system, as the write operations can terminate with no problems. Also, the variations of the throughput during the chain repair are due to the fact that read operations are processed by only 5 nodes in this period.

7.5 Metadata Experiments

In the last experiment we focus on measuring the overhead imposed by the amount of metadata exchanged in the network with ChainReaction when compare with COPS. For this purpose we simulated two datacenters in a single machine and deployed a data node for each datacenter. We resorted to this micro benchmark to overcome limitations of the COPS prototype, that was kindly made available to us by the authors, which did not support chain sizes above 1.

The two systems were also tested using the Yahoo! Cloud Serving Benchmark standard workloads. Each workload was executed by 10 YCSB client threads that submitted a total of 1,000,000 operations over 1KByte objects. In each execution we measured the size of metadata sent across datacenters and the metadata included in write operations during the first 100 seconds of execution. The values presented result from an average of the 10 clients, error bars are also displayed. We do not present results for the size of data stored at the client library as this is extremely dependent of the workload. Our solution stores one entry for each accessed data and, similarly to COPS, in the worst case, having a linear cost with the number of items read (without performing a write).

Figure 11 depicts the results for the measurements of the size of the exchanged messages (and corresponding metadata) sent across datacenters and also the size of write requests sent to the local datacenter in workload A. We can observe that COPS sends much more data across datacenters, increasing during the execution of the workload, peaking at an average of approximately 16,300 Bytes (16 KBytes). However, in our the solution the average size of one message remains stable during the workload execution and is the same for both workloads peaking at 450 Bytes (due to the usage of bloom filters). One can also observe that on COPS most of the message payload is metadata in the form of dependencies reaching 92% of the message payload. In our solution the metadata size is approximately 280 Bytes corresponding to the key, value and the two bloom filters sent in the message. Similar to the results for the propagation messages we can observe that COPS also sends larger write requests than our solution. Also the size of messages, in COPS, increases during the execution of both workloads peaking at 20,000 Bytes (19 KBytes). The behavior exposed by our system is similar to the previous scenario as the size of messages remains stable at 380 Bytes.

These results are explained by the fact that in our system, each PUT enable a client to garbage collect all locally stored dependencies, as the stabilization procedure executed by proxies before applying the PUT implicitly ensures that

those dependencies have become DC-Write-Stable(d). Additionally, in the wide-area, these dependencies are already encoded in the logical clocks used by *remote-proxies* and in the bloom filters that tag each individual PUT in remote-updates. COPS cannot rely on PUT to ensure the stability of decencies in the wide-area, leading them to present the communication overhead shown in this micro benchmark.

8. Conclusions

This paper 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 We wish to sincerely thank our shepherd Lorenzo Alvisi, the anonymous reviewers, Rodrigo Rodrigues, and Nuno Preguiça for all their comments, which were invaluable for us to improve the manuscript. Additionally, our thanks to Maria Couceiro, Amar Phanishayee, and Wyatt Lloyd for their help with bloom filters, FAWN-KV, and COPS respectively. 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] S. Almeida. Geo-replication in large scale cloud computing applications. Master's thesis, Univ. Técnica de Lisboa, 2007.

[2] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. *Comm. ACM*, 54(7):101–109, 2011.

[3] 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*, pages 223–234, 2011.

[4] N. Belarami, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX NSDI*, pages 59–72, 2006.

[5] E. Brewer. Towards robust distributed systems (abstract). In *ACM PODC*, page 7, 2000.

[6] 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*, pages 1277–1288. VLDB Endowment, 2008.

[7] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC*, pages 143–154, 2010.

[8] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *IEEE PRDC*, pages 307–313, 2009.

[9] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store for cloud computing. Technical report, CSD, Cornell University, 2011.

[10] 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*, pages 205–220, 2007.

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

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

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10:360–391, 1992.

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

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

[16] C. Lesniewski-Laas. A sybil-proof one-hop DHT. In *ACM SNS*, pages 19–24, 2008.

[17] 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*, pages 401–416, 2011.

[18] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, 2011.

[19] D. Malkhi, M. Balakrishnan, J. Davis, V. Prabhakaran, and T. Wobber. From paxos to CORFU: a flash-speed shared log. *SIGOPS OSR*, 46:47–51, Feb. 2012.

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

[21] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems (2nd Ed.)*, ACM-Press, chapter 7. Addison-Wesley, 1993.

[22] Y. Sovran, R. Power, M. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM SOSP*, pages 385–400, 2011.

[23] J. Terrace and M. J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Tech. Conference*, USA, 2009.

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

[25] R. van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, USA, 2004.