Augustus: Scalable and Robust Storage for Cloud Applications

Ricardo Padilha Fernando Pedone

University of Lugano, Switzerland *

Abstract

Cloud-scale storage applications have strict requirements. On the one hand, they require scalable throughput; on the other hand, many applications would largely benefit from strong consistency. Since these requirements are sometimes considered contradictory, the subject has split the community with one side defending scalability at any cost (the "NoSQL" side), and the other side holding on time-proven transactional storage systems (the "SQL" side). In this paper, we present Augustus, a system that aims to bridge the sides by offering low-cost transactions with strong consistency and scalable throughput. Furthermore, Augustus assumes Byzantine failures to ensure data consistency even in the most hostile environments. We evaluated Augustus with a suite of micro-benchmarks, Buzzer (a Twitter-like service), and BFT Derby (an SQL engine based on Apache Derby).

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design—Distributed Systems; C.4 [*Performance of Systems*]: Fault tolerance

Keywords Scalable storage, Byzantine fault tolerance, transactional database, cloud computing

1. Introduction

Many distributed multi-tier applications rely on a data management tier for state management. While it is simple to make stateless tiers scale and tolerate failures, the same does not hold for stateful tiers, and consequently the database is often the performance and availability bottleneck of distributed multi-tier applications. Although much effort has been put into developing sophisticated database protocols that target scalability and availability, cloud-scale computing applications, spanning a large number of servers and serving millions of simultaneous clients, often face the "database bottleneck" challenge by relying on backend storage systems that offer guarantees weaker than the traditional ACID properties (i.e., atomic and persistent transactions with strong isolation). This new generation of systems, with few or no isolation guarantees, have been termed "NoSQL," as opposed to the more traditional "SQL" systems.

The NoSQL versus SQL dilemma has given rise to an animate debate, with NoSQL advocates claiming that traditional relational databases cannot scale to "cloud environments" and SQL advocates claiming that NoSQL systems scale at the cost of pushing the complexity to the application and giving up on some of the guarantees that SQL systems offer [30]. In this paper, we argue that scalability and strong isolation at the storage level are not mutually exclusive and propose a storage system that combines a scalable design, through partitioning, with strong multi-partition isolation by means of ACID transactions. Although we are not the only ones to pursue this direction (e.g., Calvin [31], H-Store [14]), to the best of our knowledge, we are the first to consider it in the presence of arbitrary failures (i.e., Byzantine failures). As we detail later, previous scalable Byzantine fault-tolerant systems are either non-transactional (e.g., [8]) or do not offer strong multi-partition isolation (e.g., [13]).

The current generation of cloud environments offers few guarantees about fault-tolerance and availability. Providers claim that they will use "commercially reasonable efforts[1]" to prevent disruptions. Such disclaimers are necessary due to the nature of the hardware deployed in cloud environments. To keep costs low providers deploy commodity hardware, which is unreliable. Although many cloud providers do try to offer some kind of fault-tolerance at the hardware level (e.g., ECC RAM, RAID storage, redundant network links), even when these features are present, they may not be sufficient to guarantee the dependability that some mission-critical services require. For such cases, Byzantine fault-tolerance is the safest solution since it makes no assumptions about the behavior of faulty components. Moreover, this holds even in the presence of homogenous hardware since failures tend to concentrate on the same servers: more than 93% of servers that suffer a transient failure will have another incident within the year [27].

* This work was partially funded by the Hasler Foundation, Switzerland, project number 2316.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright © 2013 ACM 978-1-4503-1994-2/13/04…$15.00

[1] http://aws.amazon.com/ec2-sla/

Unfortunately, Byzantine fault-tolerant (BFT) services usually have increased latency, when compared to simple client-server interactions, and limited scalability, in the sense that adding servers does not translate into higher throughput. Much research has been done in the past years addressing the latency problem (e.g., [12, 17, 29]), and several proposed techniques could be integrated in our design. The lack of scalability is derived from the fact that BFT services rely either on state-machine replication or primary-backup replication, and neither scales. With state-machine replication every operation must be executed by every replica; thus, adding replicas does not increase throughput. With primary-backup replication, the primary executes operations first and then propagates the state changes to the backups; system throughput is determined by the primary. Although some works have proposed mechanisms to improve the throughput of BFT systems (e.g., [8, 13, 15]), Augustus, the system we designed and implemented, is the first to scale both read-only and update transactions linearly with the number of partitions.

Augustus's interface was inspired by the paradigm of *short transactions*, similar to the models proposed in [3, 6, 31]. Our definition of short transactions extends the traditional read, compare, and write operations presented in [3] with more powerful range operations. The resulting interface has proved fundamental to implement complex data management applications on top of Augustus. We implemented two such applications: Buzzer, a social network engine that provides a Twitter-like interface, and BFT Derby, a distributed relational database based on Apache Derby. While the former belongs to the NoSQL category, the latter is an SQL-based system. Both Buzzer and BFT Derby can cope with malicious clients and servers, and inherit Augustus's scalability.

This paper makes the following contributions: (1) We present a Byzantine fault-tolerant storage system based on a novel atomic commit protocol optimized for single-partition transactions and multi-partition read-only transactions. In particular, neither type of transaction requires expense signatures and thus can be executed more quickly than multi-partition update transactions. (2) We design Buzzer, a Twitter-like application that takes advantage of our storage system: Buzzer uses only cheap single-partition transactions and multi-partition read-only transactions. (3) We implement BFT Derby, a highly available SQL engine based on our extended storage interface. (4) We execute a comprehensive performance evaluation covering all of the proposed usage scenarios.

The remainder of this paper is organized as follows: Section 2 details the system model and some definitions. Section 3 introduces our scalable protocol and the novel optimizations which enable its performance. Section 4 presents the two applications built on top of it, and shows the performance evaluation results. Section 5 reviews related work, and Section 6 concludes the paper.

2. System model and definitions

We assume a message-passing distributed system with an arbitrary number of client nodes and a fixed number n of server nodes, where clients and servers are disjoint. Client and server nodes can be correct or faulty. A *correct* node follows its specification whilst a *faulty* node can present arbitrary (i.e., Byzantine) behavior. An undefined but limited number of clients can be Byzantine; the number of Byzantine servers is defined next.

One-to-one communication is through primitives *send*(m) and *receive*(m), where m is a message. If sender and receiver are correct, then every message sent is eventually received. Primitives send and receive can be implemented on top of fair links, which may fail to deliver, delay, or duplicate messages, or deliver them out of order; however, if a message is sent infinitely often to a receiver, then it is received infinitely often.

One-to-many communication is based on atomic multicast, defined by the primitives *multicast*(g, m) and *deliver*(g, m), where g is a group of servers and m is a message. Atomic multicast ensures that (a) a message multicast by a correct node to group g will be delivered by all correct servers in g; (b) if a correct server in g delivers m, then all correct servers in g deliver m; and (c) every two correct servers in g deliver messages in the same order. While several BFT protocols implement the atomic multicast properties above, we assume (and have implemented) PBFT [5], which can deliver messages in four communication steps and requires $n_g = 3f_g + 1$ servers, where f_g is the number of Byzantine servers in g and $n_g \leq n$.

We use cryptographic techniques for authentication, and digest calculation.[2] We assume that adversaries (and Byzantine nodes under their control) are computationally bound so that they are unable, with very high probability, to subvert the cryptographic techniques used. Adversaries can coordinate Byzantine nodes and delay correct nodes in order to cause the most damage to the system. Adversaries cannot, however, delay correct nodes indefinitely.

3. Scalable BFT storage

In this section, we introduce the interface provided by our storage service, describe its implementation in the absence of faulty clients and in the presence of faulty clients, present the performance optimizations that enable its good performance, and argue about the protocol's correctness.

3.1 Storage service

We consider a storage system composed of (*key*, *value*) entries, where both key and value are of arbitrary type and length. Clients access the storage by means of *short transactions* [3, 6, 31]. A client first declares all operations of a short transaction, submits it for execution and waits for its

[2] SHA-1 based HMACs and AES-128 for transport encryption.

outcome, resulting in a single round of interaction between clients and servers. Short transactions avoid the costs of client stalls and can be efficiently implemented with locks of short duration.

A transaction is composed of a sequence of operations, which can be of three classes: *comparison*, *query*, and *update* operations (see Table 1). The *cmp*(*key*,*value*) operation performs equality comparison between the value in the storage for the given *key* with the provided *value*. There are query operations to read one entry and a range of entries. The update operations allow to write the value of an existing key, insert a key to the storage, and remove a key from the storage. To execute a transaction, a server first performs all the comparison operations. If those are successful, then the server executes the query and update operations.

Class	Operation	Lock
Comparison	*cmp(key,value)*	read
Query	*read(key)*	read
	read-range(start-key, end-key)	struct read+read
Update	*insert(key,value)*	struct write+write
	write(key,value)	write
	delete(key)	struct write+write

Table 1. Transaction operations (read and write locks are acquired on single keys; structural read and write locks are acquired on partitions).

Augustus guarantees a form of strict serializability [24] that accounts for update transactions and read-only transactions submitted by correct clients.[3] In other words, we do not care about read-only transactions from misbehaving clients. For every history H representing an execution of Augustus containing committed update transactions and committed read-only transactions submitted by correct clients, there is a serial history H_s containing the same transactions such that (a) if transaction T reads an entry from transaction T' in H, T reads the same entry from T' in H_s; and (b) if T terminates before T' starts in H, then T precedes T' in H_s.

This definition ensures that every committed state of the storage can be explained by a serial arrangement of the committed transactions. Note that we do not attempt to preclude Byzantine clients from creating transactions that violate the application's integrity constraints. Handling such attacks requires access control policies and mechanisms for data recovery (e.g., maintaining database images) [16].

3.2 Storage architecture

Augustus's design is based on two strategies: *state-partitioning* and *divide-and-conquer* (see Figure 1(a)). We divide the storage entries into *partitions* and assign each partition to a group of servers (i.e., state partitioning); we handle the complexity of tolerating arbitrary failures by first rendering each server group Byzantine fault-tolerant individually, by means of state-machine replication, and then handling transaction operations across partitions with a novel BFT atomic commit protocol (i.e., divide and conquer).

The scalability of Augustus comes from the fact that atomic multicast, used to implement state-machine replication in a partition, spans the servers of the partition only. In other words, the BFT atomic commit protocol proposed by Augustus does not rely on a system-wide atomic multicast.

We divide the servers into non-intersecting groups of size $n_g < n$, out of which f_g servers can be Byzantine. To implement atomic multicast in each group, we require $n_g = 3f_g+1$ (see Section 2). We consider two schemes to distribute keys among partitions, *hashing* and *range partitioning*. All the correct servers of a group keep a complete copy of the entries of its assigned partition.

3.3 Execution in failure-free cases

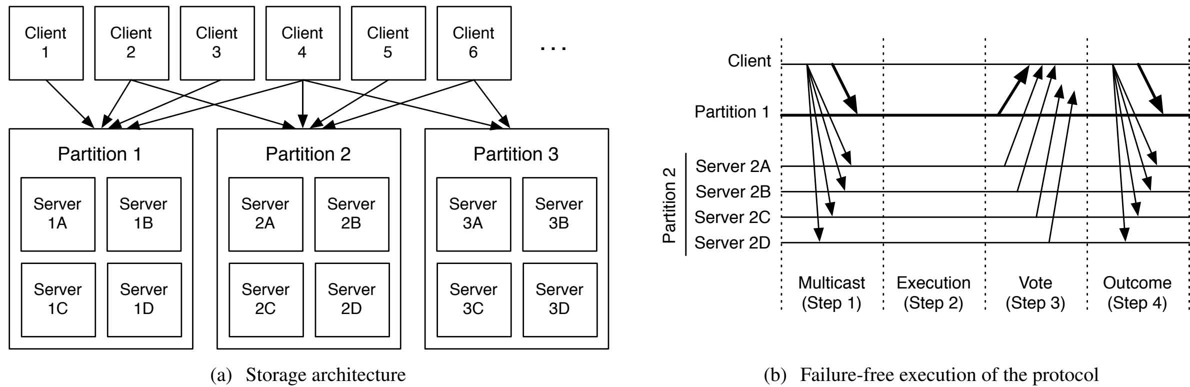
The execution of a transaction t is divided in four steps (see Figure 1(b)). In the *first step*, a correct client multicasts the operations of t to all partitions involved in t, performing one multicast call per partition. Clients determine the partitions involved in t from t's operations and the key-partitioning scheme, known by the clients. Read-range operations involve only partitions relevant to the queried range if keys are distributed using range partitioning, or all partitions if keys are distributed using hashing. Other operations involve only the partition responsible for the key in the operation.

In the *second step* of the execution, each correct server s delivers t and computes t's unique identifier from a digest of t's operations.[4] Then, s tries to acquire all of t's locks (described next). If t can be granted all its locks, s executes t's comparison operations. If one of t's locks cannot be granted or one of t's comparison operations fails, s sets t's vote to abort. Otherwise s executes t's query operations, buffers all t's update operations, and sets t's vote to commit. In either case, a server's vote on the outcome of a transaction is final and cannot be changed once cast. If s votes to commit t, t becomes *pending* at s; otherwise t is considered *terminated*. Note that within a partition, every correct server delivers the same transactions in the same order (from the properties of atomic multicast) and therefore transitions through the same sequence of states (from the deterministic procedure used to execute transactions).

A transaction can request locks on single keys or structural locks on partitions, depending on the transaction's operations (see Table 1). Compare and read operations require read locks on single keys and write operations require write locks on single keys. A read-range query requires a structural read lock on all partitions responsible for keys in the queried range,

[3] An update transaction contains at least one update operation; a read-only transaction contains only comparison and query operations.

[4] We require every two transactions to be different. It is easy for correct clients to ensure this requirement by extending transactions with a "no-op" operation containing a unique identifier.



(a) Storage architecture

(b) Failure-free execution of the protocol

Figure 1. Overview of Augustus.

	read	write	struct read	struct write
read	yes	no	yes	yes
write	no	no	yes	yes
struct read	yes	yes	yes	no
struct write	yes	yes	no	yes

Table 2. Compatibility matrix of lock types ("yes" means that the lock can be shared; "no" means that the lock is mutually exclusive).

according to the partitioning scheme, and a read lock on all existing keys within the queried range. Insert and delete operations require a structural write lock on the partition responsible for the key and a write lock on the specified key. A transaction can upgrade its read locks to update locks if the update locks do not conflict with other locks. Table 2 shows the compatibility matrix for these lock types. This scheme allows read, compare and write operations to be performed concurrently with inserts and deletes, as long as they are on different keys. Inserts and deletes on different keys can be also executed concurrently. Read-range queries are serialized with update operations to avoid phantoms [11], although they can execute concurrently with read, compare and other read-range operations. Although it is possible to implement structural locks with finer granularity, we opted for a simpler single partition-wide lock.

In the *third step* in t's lifespan, s signs its vote for t using its private key and sends to the client its signed vote and the result of t's query operations, if the vote is commit. The client collects the responses and once it gathers $f_g + 1$ matching responses from servers in g, it can determine g's vote and result. The client uses the collected votes from g to assemble a vote *certificate*, that is, $f_g + 1$ signed votes from servers in the partition that will be used to prove the partition's vote to the other partitions. The outcome of t will be *commit* if the client collects $f_g + 1$ commit votes from every partitions involved in t and *abort* otherwise.

In the *fourth step*, the client sends the certificate proving t's outcome to all servers in the involved partitions and notifies the application. When s receives a valid certificate for t from the client, s checks whether t is a pending transaction and has not been already terminated, in which case s determines the outcome of t from the certificate and proceeds accordingly: s either commits t's updates or discards them. In either case, t is no longer pending and becomes a *terminated* transaction. The locks associated with the transaction are released when the transaction terminates.

Augustus's execution model avoids deadlocks since either all the locks of a transaction are acquired atomically or the transaction aborts. If two multi-partition transactions are delivered in different order in two different partitions, then each transaction will receive at least one vote to abort from one of the involved partitions, which will lead to both transactions being aborted.

3.4 Execution under Byzantine clients

Byzantine clients can attempt to subvert the execution by trying to (a) disrupt the termination protocol and (b) abort transactions submitted by correct clients.

In the first case, a Byzantine client could (a.1) multicast non-matching operations to different partitions in the context of the same transaction or (a.2) leave a transaction unfinished in one or more partitions—notice that this may happen when a client fails by crashing. For (a.1), recall from the previous section that a transaction is uniquely identified by its operations. Thus, the non-matching operations will yield different identifiers and be considered different transactions altogether by the partitions. Since to be terminated a transaction requires a certificate, including the vote of each partition involved in the transaction, this attack will result in unfinished transactions (i.e., forever in the pending state), which is identical to case (a.2).

Our strategy to address scenario (a.2) is to rely on subsequent correct clients to complete pending transactions left unfinished. From the previous section, if a transaction t con-

flicts with a pending transaction u in some server $s \in g$, t is aborted by s. In the abort message sent by s to the client, s includes u's operations. When the client receives an abort message from $f_g + 1$ replicas in g, in addition to aborting t, the client starts the termination of u by multicasting u's operations to every partition h involved in u. If a vote request for u was not previously delivered in h (e.g., not multicast by the client that created u), then the correct members of h will proceed according to the client's request. If u's vote request was delivered in h, then correct members will return the result of the previous vote, since they cannot change their vote (i.e., votes are final). In any case, eventually the client will gather enough votes to complete pending transaction u, following the same steps as the normal case. To a certain extent, correct clients play the role of "recovery coordinators" [3].

In case (b) above, Byzantine clients can try to harm correct clients by submitting transactions that increase the likelihood of lock conflicts. This can be attempted in a number of ways. For example, by submitting transactions with many update operations or by submitting multiple transactions concurrently. Such attacks can be effective against multi-partition transactions submitted by correct clients and we cannot completely avoid them (i.e., legitimate transactions submitted by correct clients can also cause aborts). In any case, the short-duration nature of locks in Augustus limits the chances of lock conflicts. In addition to this, other measures could be used, although we do not currently implement them in our prototype, such as limiting the number of operations in a transaction or restricting the number of simultaneous pending transactions originating from a single client (e.g., [20]).

3.5 Performance optimizations

The most time-critical operation in the protocol described in the previous sections is the signing of votes, executed by each correct server of a partition, in order to compose a vote certificate for the partition. In this section we describe two optimizations that reduce this operation to multi-partition update transactions only.

3.5.1 Fast single-partition transactions

The atomic multicast protocol executed within each partition guarantees that all correct servers in the partition compute the same vote for a transaction after executing it. Therefore, the protocol presented in Section 3.3 can be optimized in two ways. First, the fourth step in the execution of a transaction is not needed; servers can terminate a transaction right after executing its operations. Second, as a consequence of abbreviated termination, servers do not have to provide signed votes to the clients, speeding up the execution of single-partition transactions. Signed votes are used to build a certificate for the transaction, used to prove the outcome of one partition to another; in single-partition transactions this is unnecessary.

3.5.2 Fast read-only transactions

As discussed in Section 3.3, the vote certificate prevents a Byzantine client from forcing the termination of a transaction in one partition without the agreement of the other partitions. Partitions participating in a multi-partition read-only transaction, however, do not need to prove the outcome of the transaction to each other; for the transaction to be serializable, each partition only needs to hold read locks until the client executes the fourth step of the protocol. For these transactions it is possible to waive the vote certificate.

Eliminating the requirement of a signed certificate allows malicious clients to create read-only transactions that observe a non-serializable execution. In Figure 2(a), transaction t_1, submitted by the Byzantine client, reads a value of x that precedes transaction t_2 and a value of y that succeeds t_2. To commit t_1 at the first partition, the Byzantine client forges the vote from partition 2, something that would be impossible if signed certificates were required.

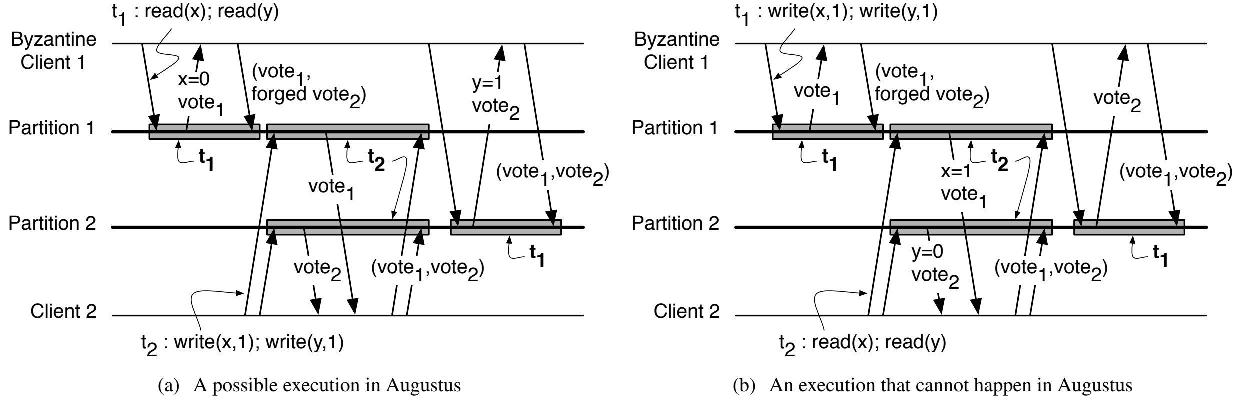
Multi-partition update transactions require a vote certificate since they must be serializable with other update transactions and read-only transactions executed by correct clients. To see why we cannot waive vote certificates from such transactions, consider the execution in Figure 2(b), where the Byzantine client uses a similar attack to commit t_1's write to key x before it sends t_1 to the second partition. As a result, t_2 reads a value of x that succeeds t_2 and a value of y that precedes t_1.

3.6 Correctness

We show that for all executions H produced by Augustus with committed update transactions and committed read-only transactions from correct clients, there is a serial history H_s with the same transactions that satisfies two properties: (a) If T reads an item that was most recently updated by T' in H (or "T reads from T'" in short), then T reads the same item from T' in H_s (i.e., H and H_s are equivalent). (b) If T commits before T' starts in H then T precedes T' in H_s.

Case 1. T and T' are single-partition transactions. If T and T' *access the same partition*, then from the protocol, one transaction executes before the other, according to the order they are delivered. If T executes first, T precedes T' in H_s, which trivially satisfies (b). It ensures (a) because it is impossible for T' to read an item from T since T' is executed after T terminates. If T and T' *access different partitions*, then neither T reads from T' nor T' reads from T and T, and T' can appear in H_s in any order to ensure (a). To guarantee (b), T precedes T' in H_s if and only if T commits before T' starts in H. In this case, recovery is never needed since atomic multicast ensures that T and T' are delivered and entirely executed by all correct servers in their partition.

Case 2. T and T' are multi-partition transactions, accessing partitions in PS (partition set) and PS', respectively. We initially consider executions without recovery.



(a) A possible execution in Augustus

(b) An execution that cannot happen in Augustus

Figure 2. Certification-free executions. Execution (a) is serializable since we do not care about t_1, a read-only transaction submitted by a Byzantine client; this execution may happen due to the fast read-only transaction optimization. Execution (b) is not serializable since t_1 precedes t_2 at partition 1 and t_2 precedes t_1 at partition 2. Augustus does not allow execution (b).

First, assume that PS and PS' intersect and $p \in PS \cap PS'$. There are two possibilities: (i) either the locks requested by T and T' are shared or (ii) at least one such lock is exclusive. In (i), property (a) is trivially ensured since neither transaction performs updates (i.e., these require exclusive locks) and thus one transaction does not read from the other; to ensure (b), T and T' appear in H_s following their termination order, if they are not concurrent. If they are concurrent, then their order in H_s does not affect property (b). In (ii), from the algorithm either (ii.a) T commits in every p before T' is executed at p or (ii.b) the other way round. This holds because otherwise T or T' or both transactions would be aborted. Without lack of generality, assume (ii.a) holds. Thus, T precedes T' in H_s. Property (a) is guaranteed because it is impossible for T to read from T' since T commits before T' is executed in p. Property (b) holds because it impossible for T' to terminate before T.

Now assume that PS and PS' do not intersect. Then T and T' can be in H_s in any order. In either case, (a) is trivially ensured. T precedes T' in H_s if and only if T commits before T' starts in H, and thus (b) is ensured. Recovery extends the lifetime of a transaction, but does not change the argument above.

Case 3: T is a single partition transaction accessing P and T' is a multi-partition transaction accessing partitions in PS'. If T is executed before T' at P, T precedes T' in H_s. Property (a) follows from the fact that T cannot read from T'; property (b) follows because T' can only finish after T. If $P \notin PS'$, then (a) trivially holds and (b) can be ensured by placing T and T' in H_s following the order they complete. Finally, if T is executed after T' at P, then T' precedes T in H_s. Property (a) holds since T' cannot read from T and it is impossible for T to commit before T'.

4. Performance evaluation

In this section, we describe the environment in which we conducted our experiments, reason about our choice of benchmarks, and experimentally assess Augustus and the applications we implemented on top of it.

4.1 Environment setup and measurements

We ran all the tests on a cluster with two types of nodes: (a) HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory, and (b) Dell SC1435 nodes equipped with two dual-core AMD Opteron processors running at 2.0 GHz and 4 GB of main memory. The HP nodes are connected to an HP ProCurve Switch 2910al-48G gigabit network switch, and the Dell nodes are connected to an HP ProCurve 2900-48G gigabit network switch. The switches are interconnected via a 20 Gbps link. The single hop latency between two machines connected to different switches is 0.17 ms for a 1KB packet. The nodes ran CentOS Linux 6.3 64-bit with kernel 2.6.32. We used the Sun Java SE Runtime 1.7.0_10 with the 64-Bit Server VM (build 23.6-b04).

Our prototype includes an atomic multicast implementation based on PBFT [5] and the transaction processing engine. The prototype was implemented in Java 7. We implemented range queries using Java's own sorted collections. In the experiments, each client is a thread performing synchronous calls to the partitions sequentially, without think time. Client processes ran on the Dell nodes. Each partition contained four servers, which ran on the HP nodes. Our largest deployment included 32 server nodes and 40 client nodes to host up to 2000 client threads, evenly distributed across nodes.

The throughput and latency numbers in all experiments were selected at the point of highest *power* in the system, that is, where the ratio of throughput divided by latency was at

its maximum. Although this does not represent the absolute maximum throughput of the system, it indicates the inflection point where the system reaches its peak throughput before latencies start to increase due to queueing effects.

4.2 Benchmark rationale

We evaluated our storage system with a series of micro-benchmarks and two different classes of applications: Buzzer, a Twitter clone that tolerates Byzantine failures, and BFT Derby, a Byzantine fault-tolerant SQL database. We detail these systems in the following sections.

Augustus transactions can be classified in four groups: (1) local update transactions, (2) local read-only transactions, (3) global update transactions, and (4) global read-only transactions, where local transactions involve a single partition and global transactions involve two or more partitions. The micro-benchmarks were designed to assess the performance of these transaction groups under various conditions, including percentage of multi-partition transactions, number of operations in a transaction, and size of data items.

Buzzer illustrates how to implement a scalable application using "cheap transactions," those belonging to the more efficient group, as assessed in the micro-benchmarks. Finally, BFT Derby shows that Augustus's interface and consistency are sufficient to serve as the storage layer of an SQL database, and boost its reliability and performance.

4.3 Micro-benchmarks

We evaluated four different types of workloads (see Table 3). Workloads A and B perform updates, while workloads C and D are read-only. The keys used in each request were picked randomly from the key space. We varied the number of partitions for all workloads from one to eight. Keys were distributed among partitions using key hashing. For the multi-partition tests, we employed six different mixes, starting with only single-partition transactions up to a 100% of multi-partition transactions. Multi-partition transactions involved two partitions.

Type	Reads (ops)	Writes (ops)	Key size (bytes)	Value size (bytes)	DB size (items)
A	4	4	4	4	3M
B	2	2	4	1K	1M
C	8	0	4	4	3M
D	4	0	4	1K	1M

Table 3. Workload types in microbenchmark.

The first observation from the experiments (see Figure 3) is that for all workloads with multi-partition transactions, Augustus's throughput increases linearly with the number of partitions. In Figure 3, for example, the topmost graph (workload A) for 0% of multi-partition transactions (leftmost cluster of bars) shows that while the throughput for one partition peaked at 40K tps, the throughput for eight partitions peaked at 320K tps, an eightfold performance increase.

The second observation from the experiments is that the overhead caused by the fourth step of the protocol, required by multi-partition transactions, depends on the size of the payload and the type of transaction, as we now explain.

In workload C, with small read-only transactions, the throughput penalty for going from single-partition to 20% multi-partition transactions with two partitions is about 33%. Increasing multi-partition transactions to 40% causes a 45% throughput penalty, while 60% of multi-partition transactions cause a 52% penalty. With 100% of multi-partition transactions, there is a 66% reduction in throughput. In workload A, with small update transactions, the 20% multi-partition transactions penalty with two partitions is 50%, i.e, from 76K tps to 38K tps. A multi-partition mix of 40% causes a 60% penalty, and a 100% multi-partition mix causes a 72% throughput penalty, which then peaks at 21K tps.

Comparatively, in workloads B and D, with transactions with large payload, there is almost no performance loss until 60% multi-partition transactions. In workload D, with large read-only transactions, the throughput penalty for going from 0% to 60% multi-partition transactions with two partitions is about 5%. At 100% multi-partition transactions, the penalty is about 15%, and throughput peaks at 17K tps. Workload B, with large update transactions, the reduction in throughput for going from single-partition to 60% multi-partition transactions is about 18%. At 100% multi-partition transactions, the decrease in throughput is about 37%.

The difference in behavior between small and large payloads is explained by CPU and network usage. Large payloads coupled with small percentages of multi-partition transactions lead to saturation of network links. We observed peaks of 19 Gbps network traffic at the interconnect between switches in workload D. The saturation of the network links is partially due to the message authentication method employed (shared secret key HMACs) and to the very nature of PBFT, which requires at least $2f+1$ replies for each request. Larger percentages of multi-partition transactions lead to CPU saturation, particularly in the case of small transactions, due to the necessity of signing the commit certificates. Workloads A and C, for example, saturate servers CPUs long before the saturation of network links.

We conclude that Augustus's transactions perform as follow: (1) Local read-only transactions have the best performance because their termination is optimized and there is minimal overhead to acquire read locks. (2) Local update transactions come next: although their termination is optimized, the acquisition of the exclusive write locks is a slightly more expensive operation in our implementation. (3) Global read-only transactions are second to last, since they incur the overhead of the multi-partition termination protocol but do not require the computationally expensive vote certificate. (4) Global update transactions have the worst performance: their termination requires the complete termination protocol and the computationally expensive signed vote certificate.

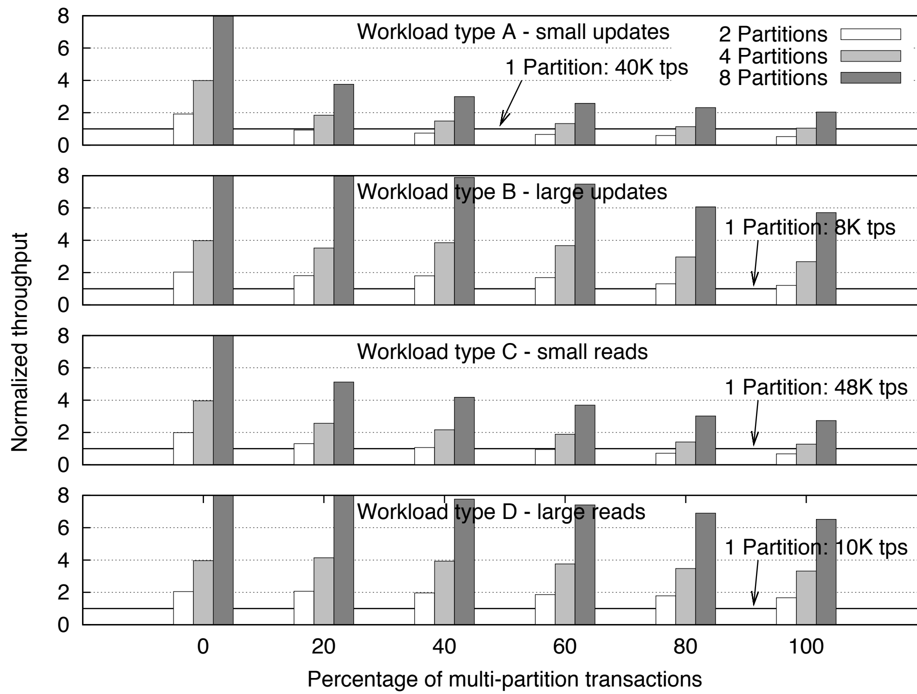


Figure 3. Normalized throughput versus percentage of multi-partition transactions.

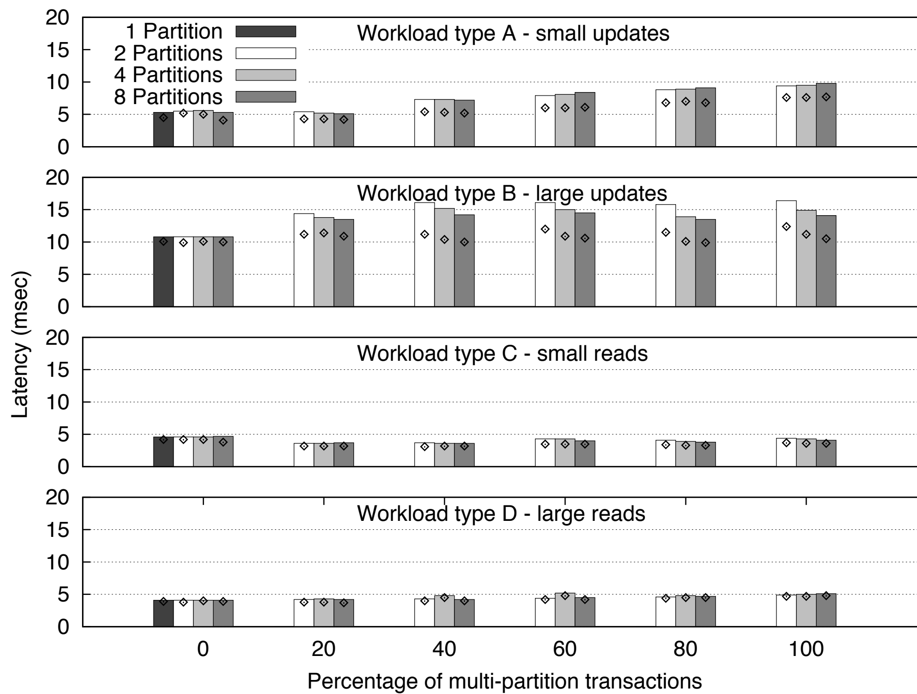


Figure 4. Latency of multi-partition transactions (bars show the 95-th percentile of latency and diamonds in bars the average).

In our experiments, read-only workloads have a lower latency than the equivalent-sized update workloads (see Figure 4). In both workloads C and D the average latencies were below 5 ms across all mixes; moreover, the latencies of multi-partition read-only transactions were mostly unaffected by the percentage of multi-partition transactions. This confirms our expectations from the optimized termination protocol for read-only multi-partition transactions. Latencies of multi-partition updates are higher than the single-partition updates, and increase with the percentage of multi-partition transactions. Workload A starts with average latencies between 4.1 and 5.2 ms, and ends at 7.7 ms for 100% multi-partition transactions. Workload B has its average latency between 10.5 and 12.4 ms for update transactions.

Type	Multi-partition (%)	Number of partitions 2	4	8
A	20	0.26	0.42	2.02
	40	0.42	0.56	2.41
	60	0.54	0.72	2.80
	80	0.77	0.96	2.94
	100	0.87	1.13	3.28
B	20	0.15	0.30	0.96
	40	0.28	0.42	1.36
	60	0.40	0.54	1.63
	80	0.56	0.79	1.98
	100	0.66	0.80	2.16

Table 4. Maximum abort rates (in %) in workloads A and B.

Abort rates for the micro-benchmarks were very low, most of the time under 1% (see Table 4). The worst value, 3.28%, was observed for workload A running under a 100% multi-partition mix on eight partitions. Types C and D had no aborts, by virtue of being read-only.

We now consider the effects of Byzantine clients on the execution. Byzantine clients can have a negative impact on performance by leaving unfinished transactions in the system. We evaluate the impact of Byzantine clients in workloads A and C with four partitions and mixes of 20, 60, and 100% of multi-partition transactions. In these experiments, the total number of clients is fixed (320) and we vary the percentage of Byzantine clients in the workload from zero up to 32%. While correct clients follow the workload mix of local and global transactions, Byzantine clients always submit multi-partition transactions only and never terminate them. This behavior corresponds to attack (a.2), discussed in Section 3.4. Byzantine clients were allowed to submit only one transaction at a time [20].

Overall, throughput decreases more or less proportionally with the increase of Byzantine clients (see Figure 5). This happens for two reasons. First, Byzantine clients leave all their transactions pending and so the higher the proportion of Byzantine clients in the workload, the fewer transactions are committed. Second, pending transactions left by Byzantine clients must be finished by correct clients, which increases the amount of work that correct clients must do, a fact that has also an impact on latency.

Although latency of committed transactions increases with the number of Byzantine clients, this effect is more noticeable with small percentages of global transactions (see Figure 5). For workload A with 20% of globals, the average latency increases from 4.03 ms to 5.45 ms. For workload C, the average latency goes from 3.09 ms to 4.25 ms. The relatively small increase in latency for workload A is explained by the fact that the overhead of the recovery protocol is relatively insignificant if compared to the cost of signing the votes. This is particularly visible when all transactions are multi-partition: average latency increases only from 7.24 ms to 7.49 ms. For workload C, where there is no need for signed commit certificates, the average latency goes from 4.73 ms to 5.35 ms in the mix with 100% of global transactions.

4.4 Buzzer benchmark

Buzzer is an application developed on top of Augustus that implements an API similar to Twitter. It contains methods to: (a) create an account, (b) follow and un-follow users, (c) find out who follows an user, (d) post new messages, and (e) retrieve a user's *timeline*, i.e., the most recent messages sent by users being followed.

We compared our system to Retwis,[5] another Twitter "clone." Retwis implements the same API as Buzzer, but relies on the Redis key-value store.[6] Redis is well-known for its performance, but does not offer Byzantine fault-tolerance or multi-partition transactions. In other words, Redis key space can be partitioned, but it is up to the application to ensure multi-partition consistency. Our comparison was based on the three most frequent operations on social networks: posting a new message, following a user, and retrieving a user's timeline.

Retwis implements these operations as follows. Posts are first-level entities, with their own unique key. Each user has a list of friends (i.e., people they follow), a list of followers (i.e., people who follow them), a list of posts that belongs to them, and a list of posts that is their timeline. Following a user is a two-command operation that adds one user to the list of followers of another user, and inversely adds the other to the list of friends of the first. A new post requires the acquisition of an unique PostId (a shared atomic counter), to avoid duplicate entries, and the submission of the post data linked to the PostId. The PostId is then added to the user's post list, and subsequently to the timelines of all the followers of that user. Retrieving the timeline first fetches the timeline list, then retrieves the posts by their ids. As explained below, the acquisition of a PostId imposes a severe performance penalty on Retwis. We opted not to change the implementation of Retwis and benchmark it as is for two reasons: (1) changing the behavior of Retwis would defeat

[5] http://retwis.antirez.com/

[6] http://redis.io/

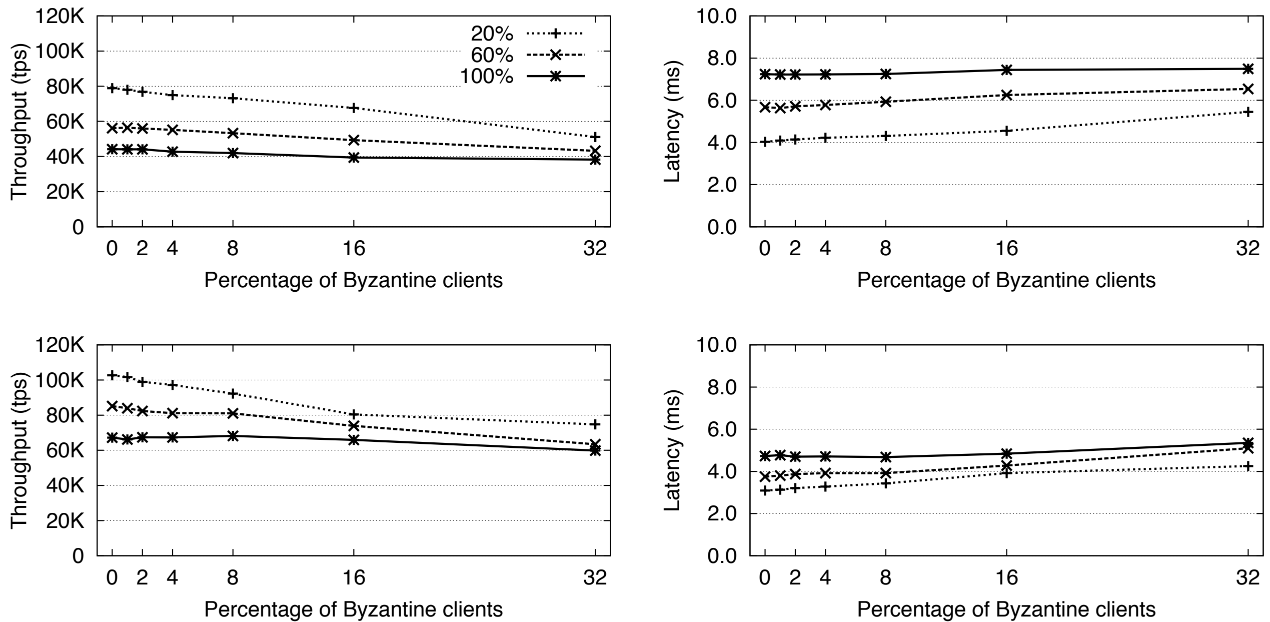


Figure 5. Impact of Byzantine clients on throughput (left) and latency (right) in workloads A (top) and C (bottom).

the goal of having a public, well-known reference benchmark; (2) Retwis is a good representative of the architecture that is used in this type of social networks sites.[7]

We implemented Buzzer's primitives as follows. Posts are linked to the user who posted them. Posting a new message is a simple insert operation where the key is a composition of the user's id and a unique timestamp. The lists of friends and followers are also user-prefixed. Keys are distributed among partitions using range partitioning. Range boundaries between partitions are defined such that user accounts and posts are evenly distributed across partitions, but the data of a single user is kept together in a partition. Due to this distribution, creating an account and posting a new message are single-partition transactions. This enables efficient range queries because these will be restricted to single-partition read-only transactions for any given user. Following an user is a two-insert transaction, one for the friends list and one for the user's followers list. This transaction is, at worst, a two-partition update. Retrieving the timeline requires fetching the list of friends and then performing a multi-partition range query for the posts of the friends. The user-centric range partitioning scheme enables efficient range queries because all read-range operations will be restricted to a single partition when obtaining the data for a given user.

The major difference between Retwis and Buzzer is the way the timelines are stored and compiled. In Retwis, timelines are precomputed since updates are always local and are no more expensive than read-only operations. In Buzzer, we implemented timeline operations using global read-only transactions. This design choice has been advocated by [28] as the better option for systems that need to deal with a high rate of new messages. It is also important to notice that since Augusts guarantees strict serializable executions, any causal dependencies between posts will be seen in the correct order. More precisely, if user B posts a message after receiving a message posted by user A, no user who follows A and B will see B's message before seeing A's message. In Retwis, this scenario may result in different timelines for different users.

The social network data used in the benchmarks contained 100,000 users, and the connections were generated using a Zipf distribution with a size of 20 and a skew of 1, shifted by one.[8] In other words, each user followed at least one other user, 50% of users followed up to four other users, and 95% of users followed up to 17 other users. For the "Post" and "Follow" benchmarks, we started with an empty data store, i.e., there was no relationship data between the different users, and therefore no timelines to update. For the "Timeline" benchmark, we preloaded the same randomly generated social network. The "Mix" benchmark performed all of the operations from the previous benchmarks using the same preloaded network, with the following composition:

[7] Tumblr, a Twitter competitor, assigns unique IDs to posts to build users' inboxes: http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html

[8] The publicly available statistics on the Twitter network seem to fit Zipf's law reasonably well. See http://www.sysomos.com/insidetwitter/ for a follower/following breakdown.

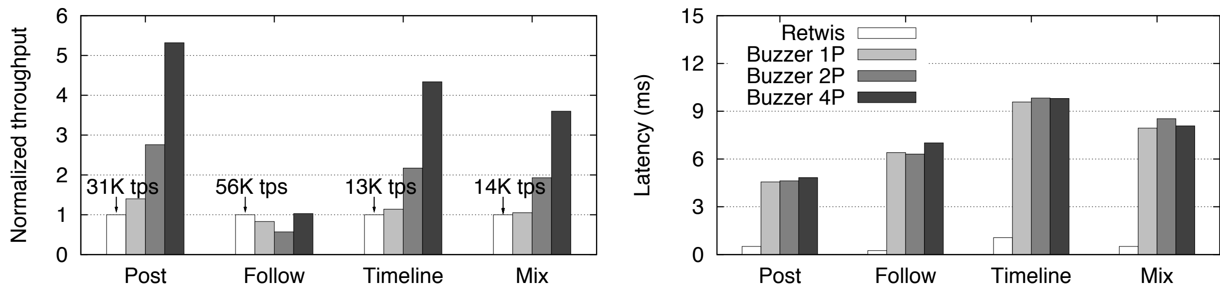


Figure 6. Buzzer's throughput and latency. Throughput is normalized by the corresponding Retwis performance. Buzzer-xP series represent different number of partitions. The baseline uses a single instance of Redis.

85% of calls retrieved timelines, 7.5% posted new messages, and the last 7.5% were requests to follow other users.

The fact that Retwis needs to acquire a PostID for each new post causes a bottleneck that explains our better throughput even with a single partition (see Figure 6). Since posts are a single-partition update transaction they scale linearly with the number of partitions. In these executions, the atomic multicast is the largest component of the overall latency, and it remains so regardless of the number of partitions.

Follow operations show a reverse picture, where the Retwis instance overcomes Buzzer. The throughput loss from one to two partitions is due to the use of signed vote certificates, which are computationally expensive to generate and verify. However, Buzzer's performance improves with 4 partitions, doubling the throughput of the two partition scenario and slightly overtaking the baseline at 57k tps. Latencies for this scenario are high, and due mainly to the creation and verification of the votes and certificates, but also due to the extra round of messages required to terminate the transactions.

The Timeline benchmark shows that our decision to implement the timeline feature using a different approach than Retwis payed off already for a single partition. The Mix benchmark performed as expected, following the numbers of the Timeline benchmark closely, but with an abort rate of 15%. This abort rate is consistent across all tested configurations, and explained by our locking mechanism, which prevents concurrent inserts and range queries (see Section 3.3).

With an exception for the single-partition scenario, latency for the Timeline benchmark is very similar for all scenarios, and here again the atomic multicast takes the largest share of the latency. Retwis latencies are compatible with normal values for a single, non-replicated server.

4.5 BFT Derby benchmark

We illustrate Augustus's expressive interface and strong consistency with BFT Derby, a highly available SQL database server. Moreover, thanks to Augustus's capability to tolerate Byzantine failures, BFT Derby can tolerate arbitrary behavior of its storage components. We chose the Apache Derby server as the SQL front-end for BFT Derby because it is implemented in Java and has a modular design, which supports the integration of different subsystems, including storage. We implemented a persistence provider and a transaction controller for Derby supporting a subset of the SQL language to replace the default implementations.

Our prototype is simple, however it is enough to demonstrate how Augustus can be used to boost the performance of an SQL database designed to run as a standalone server. BFT Derby currently implements single-statement SQL transactions for INSERT, UPDATE, DELETE, and SELECT operations, and multi-statement SQL transactions without range-based SELECT statements. Single-statement transactions are translated directly into Augustus transactions. Multi-statement transactions rely on a session-unique transaction cache. SELECT statements are immediately executed as single Augustus read-only transactions, and their results are preserved in the transaction cache. Update statements (e.g., INSERTs, UPDATEs, and DELETEs) are batched in the transaction cache and dispatched as a single Augustus's update transaction on COMMIT. The previously read items in the transaction cache are included in the COMMIT transaction as *cmp* operations. Therefore, the updates only take place if all previously read items are still up to date.

Since transaction isolation and atomicity are provided by Augustus, multiple Derby instances can be deployed against the same storage, each instance operating as an individual client of the storage system (see Figure 1). Moreover, relying on Augustus for isolation allows us to disable the concurrency control in the SQL server, which can run under a lower and more efficient isolation level (i.e., READ COMMITTED).

Bypassing Derby's internal persistence and isolation mechanisms comes at a cost. When using its internal mechanisms, Derby tracks changes to optimize caching and updates on indexes, and provides interfaces for third-party providers to notify its caching and indexing subsystems of changes. When running several Derby instances on top of the same storage backend, it is not safe to perform caching unless the backend

functionality is extended to support cache lifecycle management. We did not implement this Derby extension; instead, we disabled Derby's cache mechanism when deploying multiple instances of the database on top of Augustus.

All experiments were performed against a single table containing one integer primary key and an integer data field. We benchmarked single-statement transactions: INSERT, UPDATE, SELECT on the primary key and SELECT on a range. These statements were performed as complete transactions, i.e., a COMMIT was issued after each statement. Keys were distributed among partitions using range partitioning. INSERTs were executed against an empty database, and UPDATE and SELECTs were tested using the dataset from the workload A of the micro-benchmark. We established a baseline by measuring all scenarios on a standalone instance of Derby. The standalone instance used the default settings for cache and index optimization. All measurements were performed against in-memory databases.

We deployed BFT Derby using a "three tier"-like architecture: we used separate nodes to host the clients, the Derby front-ends, and Augustus servers. To assess the scalability of our implementation, we varied the number of Derby front-ends running on top of a single Augustus partition, and then we doubled the number of partitions. The standalone Derby instance (our baseline) was deployed on a different machine than the clients.

In the "Insert" benchmark, we observed that in a single partition BFT Derby performs inserts as fast as the baseline: both Derby baseline and BFT Derby peak around 13K tps, for a single partition with one front-end (see "1P1F" in Figure 7). When a second SQL front-end server ("1P2F") was added, throughput peaked at 29K tps, twice the single front-end throughput. A third SQL front-end ("1P3F") does not result in a proportional increase in throughput since the single partition reaches its maximum performance. Four front-ends deployed with two Augustus's partitions ("2P4F"), however, reach a throughput of 61K tps, four times the throughput of a single front-end. BFT Derby latencies remained approximately the same for all scenarios and higher than Derby baseline because of the underlying BFT atomic multicast.

The "Select" benchmark presented a slightly different picture. In this case, BFT Derby had a 36% throughput drop from the baseline. The Derby baseline peaked at 22K tps, while BFT Derby peaked at 14K tps. This is explained by Derby's superior caching and internal indexing mechanisms. Using a second SQL front-end increased throughput to 29K tps, twice as much as the single front-end case. A third SQL front-end only increased throughput to 40K tps. Adding two partitions with four front-ends resulted in a peak throughput of 65K tps. Latencies behaved the same as the Insert benchmark.

The "Range" benchmark measured the throughput of a SELECT statement querying a range of keys. The peak throughput for a single partition is about 60% of the 25K tps of the baseline. Adding a second SQL front-end doubles the throughput and brings it to 30K tps, and the third front-end brought it to 39K tps. As expected, these values follow the single SELECT benchmark closely. The introduction of a second partition pushed throughput over 60k tps. This is explained by the fact that using range partitioning allows the read-range operations to be executed as single-partition transactions, and thus scale linearly.

For the "Update" benchmark, BFT Derby with a single front-end performed at about the same level as the baseline; 11K tps versus 12K tps at peak performance, respectively. The second SQL front-end doubled throughput, bringing it to 23K tps. The third SQL front-end only increased throughput to 27K tps. With four front-ends, throughput peaked at 45K tps. One could expect updates to perform just as well as inserts since in both cases statements can be mapped directly to Augustus transactions. However, for each update statement Derby internally issues the equivalent of a SELECT statement to make sure that the entry being updated exists. Despite the fact that Augustus can handle the UPDATE invocation correctly, i.e., to update the entry only if it already exists, we could not override this behavior inside Derby and disable it. As a result, each UPDATE statement was translated to two Augustus transactions: one read transaction to make sure the key existed, and then a compare and write transaction at commit. This explains the higher latency, which is the double of other scenarios.

5. Related work

Although the literature on Byzantine-fault tolerant systems is vast, until recently most research was theoretical, within the distributed computing community (e.g., [4, 19, 21, 25]). After Castro and Liskov's PBFT showed that tolerating Byzantine failures can be practical [5], many papers proposed techniques to reduce the latency of BFT state-machine replication (e.g., [1, 7, 12, 17, 29]) and tolerate malicious attacks in database systems [32]. We focus next on previous work related to improving throughput under Byzantine failures, the main issue addressed by Augustus. Existing protocols can be divided into those that do not support transactions (e.g., [2, 8, 13, 18]), more suitable for file system replication, and those with transaction support (e.g., [9, 23, 26]).

Farsite [2] is one of the first storage systems to explore scalability in the context of Byzantine failures. It provides BFT support for the metadata of a distributed filesystem, and confidentiality and availability for the data itself, through encryption and replication. Differently from Augustus, Farsite is not intended for database applications, as it does not support transactions and assumes low concurrency and small-scale read-write sharing.

Although the separation of agreement and execution is presented in [33], it is only in [8] that we see the first application of the reduction to an $f + 1$ execution quorum coupled with on-demand replica consistency. While the approach does improve throughput, the improvement depends on the num-

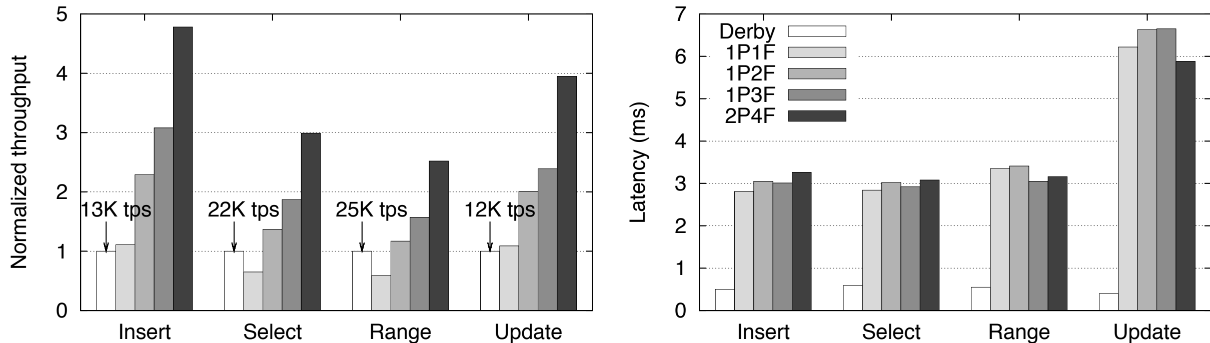


Figure 7. BFT Derby throughput and latency. BFT Derby throughput is normalized by the corresponding Derby throughput. Each xPyF series represents the number of partitions x and the number of SQL front-ends y used in the measurements.

ber of "cross-border" (i.e., multi-partition) requests, which require an expensive inter-partition state-transfer protocol.

Zzyzx [13] implements a centralized locking system that allows clients, once they acquire the required locks, to submit an unlimited number of sequential operations to a BFT quorum of log servers. BFT agreement is only executed at the end to ensure the client submitted the same operations in the same order to all log servers. Scalability is achieved by dividing the application state across several partitions of log servers. As pointed out by the authors, the solution is optimized for the specific single-client, low-contention, single-partition data scenario. It is also important to point out that throughput of the whole system is bound by the throughput of the centralized locking system.

Oceanstore [18] separates replicas in tiers and introduces delayed dissemination of updates. Although it supports some form of transactions, it is based on an update dissemination protocol that requires a complex model of replicas containing both *tentative* and *commited* data. If strong consistency is required, the system executes as fast as the primary tier.

Early work on BFT databases by Garcia-Molina et al. [10] assumed serial transaction execution and did not target throughput scalability. Particularly related to our proposal is the commit protocol proposed by Mohan et al. [22] for terminating distributed transactions under Byzantine failures. Augustus differs from this protocol in that each transaction participant (i.e., a partition) can tolerate Byzantine failures and atomic commit is executed among correct participants. In [22], the transaction coordinator collects votes from the participants and all transaction members run a BFT agreement protocol to decide on the transaction's outcome.

Recent protocols that provide Byzantine fault-tolerance in the context of databases are presented in [9, 23, 26]. In [9] and [26], transactions are first executed at one replica and then propagated to the other replicas, where they are checked for consistency and possibly commit. While the protocol in [9] guarantees snapshot isolation and relies on broadcasting transactions twice (i.e., in the beginning of the transaction, to assign it a consistent snapshot, and at the end to certify the transaction), the protocol in [26] ensured serializability and broadcasts transactions at the end of their execution. These approaches support single-partition transactions and do not scale throughput under update transactions. In [23] we briefly discuss the use of a BFT atomic commit protocol to terminate multi-partition transactions, although without Augustus's optimizations and with no in-depth evaluation.

In [15], Kapritsos and Junqueira propose a way to improve the scalability of agreement protocols (i.e., the message ordering). As the authors indicate themselves, scalability of the agreement protocol does not imply scalability of the execution of requests. Their work is orthogonal to ours, since our model is agnostic to the underlying agreement protocol, and could use implementations more scalable than PBFT.

6. Final remarks

Scalability, strong consistency, and Byzantine fault-tolerance appear to be conflicting goals when the topic is cloud-scale applications. Augustus approaches this issue with an optimized BFT atomic commit protocol that enables scalable throughput under read-only and update transactions. Moreover, single-partition transactions (both read-only and updates) and multi-partition read-only transactions are efficiently executed (i.e., without the need of expensive signed certificates).

We developed two scalable applications on top of Augustus, thereby demonstrating its functionality. Buzzer shows how common operations in a social network application can benefit from Augustus efficient transactions. BFT Derby illustrates how an existing centralized databases can be made scalable under common SQL statements. Although our prototype is simple (i.e., it currently only supports single-statement transactions with a SELECT, INSERT, UPDATE and DELETE operations and multi-statement transactions on single keys), it is the foundation for more complex SQL-based data management systems.

Acknowledgments

We wish to thank Roxana Geambasu, our shepherd, and the anonymous reviewers for the insightful comments that helped improve the paper.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[4] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, 1993.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[6] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX*, 2012.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, 2006.

[8] T. Distler and R. Kapitza. Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. In *EUROSYS*, 2011.

[9] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault-tolerant database replication. In *EUROSYS*, 2011.

[10] H. Garcia-Molina, F. Pittelli, and S. Davidson. Applications of byzantine agreement in database systems. *ACM Transactions on Database Systems*, 11(1), 1986.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. In *EUROSYS*, 2010.

[13] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *DSN*, 2010.

[14] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[15] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *HotDep*, 2010.

[16] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, 2005.

[17] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4), 2009.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[19] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[20] B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. *ICDCS*, 2006.

[21] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[22] C. Mohan, H. R. Strong, and S. J. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *PODC*, 1983.

[23] R. Padilha and F. Pedone. Scalable byzantine fault-tolerant storage. In *HotDep*, 2011.

[24] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[25] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), 1980.

[26] F. Pedone and N. Schiper. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society*, 18(1), 2012.

[27] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. In *SIGMETRICS*, 2009.

[28] A. Silberstein, J. Terrace, B. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, 2010.

[29] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, 2009.

[30] M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4), 2010.

[31] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[32] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.

[33] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, 2003.