Callinicos: Robust Transactional Storage for Distributed Data Structures

Ricardo Padilha, Enrique Fynn, Robert Soulé, and Fernando Pedone,
Universitá della Svizzera Italiana (USI)

https://www.usenix.org/conference/atc16/technical-sessions/presentation/padilha

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Callinicos: Robust Transactional Storage for Distributed Data Structures

Ricardo Padilha Enrique Fynn Robert Soulé Fernando Pedone
Universitá della Svizzera italiana (USI)
Switzerland

Abstract

This paper presents Callinicos, a robust storage system with a novel transaction protocol that generalizes mini-transactions. This protocol allows Callinicos to cope with Byzantine failures, support cross-partition communication with transactions, and implement on-demand contention management. We have evaluated Callinicos with a set of micro-benchmarks, and two realistic applications: a Twitter-like social network and a distributed message queue. Our experiments show that: (i) cross-partition communication improves performance by reducing the number of aborts, and (ii) the conflict resolution protocol results in no aborts in the presence of contention and no overhead in the absence of contention.

1 Introduction

Many application domains including retail, healthcare, and finance, have a need for storage systems that tolerate failures and scale performance without sacrificing consistency. However, designing systems that satisfy these requirements is a daunting task. Among the various approaches proposed in recent years, mini-transactions strike an attractive balance between functionality and performance [2]. Mini-transactions allow applications to atomically access and conditionally modify distributed data. They are also amenable to optimizations that result in scalable performance, such as update batching, message piggybacking, and state partitioning.

Like database transactions, mini-transactions hide the complexities that result from concurrent execution and failures. However, mini-transactions optimize the execution and commit of transactions by exposing a restricted set of operations. Thes operations allow a transaction to read a storage entry; compare an entry with a value (i.e., equality comparison); and update the value of an entry. A mini-transaction is only committed if all its compare operations are successful. As a consequence of these restrictions, mini-transactions can piggyback the last transaction action onto the first phase of the two-phase commit, saving a network round-trip. Despite their simple execution model, several non-trivial applications have been developed with mini-transactions, including a cluster file system and a group communication service.

Yet, despite the fact that several storage systems from the research community have proposed the use of mini-transactions [2, 34, 35], few real-world applications have employed them. This paper argues that mini-transactions suffer from several problems that have hindered their wide-spread adoption: (i) they provide only limited reliability, (ii) they disallow indirect memory accesses and prevent data from transferring from one partition to another within a transaction, and (iii) for workloads with high-contention (e.g., hot spots), they are affected by frequent aborts which leads to increased latencies. Below, we discuss each of these issues in more detail.

Limited reliability. Although mini-transactions can tolerate benign failures, such as crash failures, there are many other types of failures that occur in data centers. Byzantine failures are a common occurrence [16], resulting from any number of reasons including disk faults [8, 38], file system bugs [50], or human errors [23]. Mission-critical services require strong end-to-end guarantees that hold all the way up to the application layer.

Restricted operations. To provide scalable performance, mini-transactions disallow indirect memory accesses and prevent data from transferring from one partition to another within a single transaction. This restriction significantly impacts how applications can implement many common, but surprisingly simple operations. As a motivating example, consider the pseudocode to atomically swap the contents of two stored keys shown in Figure 1 (left). The operations could not be implemented within a single mini-transaction, since key_a and key_b may be stored in different partitions and there is no way for partitions to exchange information within a transaction. The only way one could implement the above code would

```
$\textbf{swap\_tx}(key_a, key_b)$ {
    $x \leftarrow read(key_a)$
    $y \leftarrow read(key_b)$
    $write(key_a, y)$
    $write(key_b, x)$
}
```

```
$\textbf{swap\_tx1}(key_a, key_b)$ {
    $x \leftarrow read(key_a)$
    $y \leftarrow read(key_b)$
} // return x,y
$\textbf{swap\_tx2}(key_a, key_b, x, y)$ {
    $cmp(key_a, x)$
    $cmp(key_b, y)$
    $write(key_a, x)$
    $write(key_b, y)$
}
```

Figure 1: A transaction to swap two entries (left) and its schematic implementation using mini-transactions (right).

be to split the *swap* operation in two independent transactions, *swap_tx1* that reads both keys and *swap_tx2* that updates them. However, with two transactions, the *swap* operation would no longer be atomic and a third transaction could execute between *swap_tx1* and *swap_tx2*. This third transaction could change either of the values, which would lead to a non-serializable execution. One could extend *swap_tx2* with the mini-transaction compare operation (*cmp*) to conditionally write the new values if they have not been modified, and abort otherwise, as shown in Figure 1 (right). Under high-contention workloads, such an implementation would lead to frequent aborts, decreasing storage throughput and increasing latency.

Contention. Mini-transactions rely on optimistic concurrency control (e.g., [12, 25, 27, 40, 41]). Unfortunately, in the presence of contention, optimistic concurrency control leads to aborts, poor use of system resources and increased latencies. It also put additional burden on developers, since key-value stores with optimistic concurrency control often delegate the task of managing contention to application designers.

This paper presents Callinicos, a storage system designed to address the limitations of mini-transactions. While prior work has addressed some of the above challenges individually, no system at present addresses all of them (see Table 1). Augustus [35] augments mini-transactions with support for Byzantine fault tolerance. Just like Sinfonia's mini-transactions [2], Augustus's transactions are prone to aborts in the presence of contention. Calvin [44], Rococo [34], and H-Store [43] account for contention at different degrees, as we detail in § 6, but none of these systems can cope with Byzantine failures. In contrast, Callinicos tolerates the most extreme types of failure, is not constrained by restricted operations, and never aborts transactions due to contention.

Callinicos implements *armored-transactions*, a novel transaction model that can split the execution of an armored-transaction in rounds to cope with data contention and allow partitions to exchange data. Armored-transactions execute in a single round, like mini-transactions, if they are not subject to data contention

		Contention management	
		Without	With
Failure Model	Benign	Sinfonia [2]	Rococo [34] H-Store [43] Calvin [44]
	Byzantine	Augustus [35]	**Callinicos**

Table 1: Overview of transactional key-value stores.

and cross-partition communication. Additional rounds are used to order armored-transactions that conflict and exchange data across partitions within a single and atomic armored-transaction. Armored-transactions can be viewed as a generalization of basic mini-transactions, which allow for cross-partition communication, and implement on-demand contention management.

Using Callinicos, we have implemented two real-world applications: *Buzzer*, a twitter clone, and *Kassia* a distributed message queue based on Apache Kafka. Although it would have been possible to implement these applications using mini-transactions, their design would be significantly more complicated and the performance would be poor, as we show in § 5. We have evaluated Callinicos under a variety of deployment and workload scenarios, and our experiments show that in the absence of contention, Callinicos introduces no overhead in the execution of transactions and can scale throughput with the number of nodes; in the presence of contention it orders transactions to avoid aborts. Overall, this paper makes the following contributions:

- It details a novel multi-round transactional execution model that generalizes the mini-transaction model, while allowing for cross-partition data exchange and contention management.
- It describes the design and implementation of a robust, distributed storage system built using the new transactional execution model.
- It demonstrates through experimental evaluation that the transactional model and storage system offer significant performance improvements over mini-transactions.

The remainder of the paper is structured as follows: it details the system model (§ 2), discusses Callinicos's design (§ 3), and presents the transaction execution protocol (§ 4). Then it evaluates the performance (§ 5), reviews related work (§ 6), and concludes (§ 7).

2 System model and definitions

We consider a distributed system with processes that communicate by message passing. Processes do not have access to a shared memory or a global clock. The system is asynchronous (i.e., no bounds on processing times or message delays) and there is an arbitrary number of clients and a fixed number n of servers, where clients and servers are disjoint.

Processes can be *correct* or *faulty*. A correct process follows its specification; a faulty, or Byzantine, process presents arbitrary behavior. There is a bounded although arbitrary number of faulty clients. Servers are divided into disjoint groups. Each group g contains n_g servers, out of which f_g can be faulty.

Processes communicate using either one-to-one or one-to-many communication. One-to-one communication guarantees that if sender and receiver are correct, then every message sent is eventually received. One-to-many communication is based on atomic multicast and ensures that: (a) a message multicast by a correct process to group g will be delivered by all correct processes in g; (b) if a correct process in g delivers m, then all correct processes in g deliver m; and (c) every two correct processes in g deliver messages in the same order.

Atomic multicast algorithms need additional assumptions in the system model (e.g., partial synchrony [21]). These assumptions are not explicitly used by our protocols. While several BFT protocols implement the atomic multicast properties enumerated above (e.g., [10, 26]), we assume (and have implemented) PBFT [14], which can deliver messages in four communication steps and requires $n_g = 3f_g + 1$ servers.

We use SHA-1 based HMACs for authentication, and AES-128 for transport encryption. We assume that adversaries (and faulty processes under their control) are computationally bound and unable, with very high probability, to subvert the cryptographic techniques used. Adversaries can coordinate faulty processes and delay correct processes in order to cause the most damage to the system. Adversaries cannot, however, delay correct processes indefinitely.

3 Storage Design

Callinicos is a distributed key-value store with support for transactions. Like other systems that implement mini-transactions, Callinicos uses state partitioning to improve scalability. While the partitioning algorithm has an impact on performance, the choice of partitioning scheme is orthogonal to the design of Callinicos.

Clients have access to a *partition oracle*, which knows the partitioning scheme used for a particular deployment. The partition oracle can be implemented using a variety of designs: centralized, replicated, fully-distributed, etc. Each design has different tradeoffs for performance overhead on the system. Our Callinicos prototype uses static partitioning and each client has a-priori knowledge of the partitioning scheme.

$l \in Lit$	*Literals*
$k \in \mathscr{K}$	*Keys*
$v \in ID$	*Identifiers*
$t ::= s$	Transaction
$s ::= s_1;s_2$	Sequence
$\mid v = e$	Assignment
$\mid \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$	Conditional Branch
$\mid \mathsf{while}\ e\ \mathsf{do}\ s$	While Loop
$\mid r \mid w \mid c$	Transaction Operators
$e ::= e_1 \&\& e_2 \mid e_1 \vert\vert e_2 \mid !\, e_1$	Logical Expr.
$\mid e_1 > e_2 \mid e_1 < e_2 \mid e_1 == e_2$	Relational Expr.
$\mid e_1 >= e_2 \mid e_1 <= e_2 \mid e_1 != e_2$	
$\mid e_1 * e_2 \mid e_1 / e_2$	Multiplicative Expr.
$\mid e_1 + e_2 \mid e_1 - e_2$	Additive Expr.
$\mid v \mid l$	Variable or Literal
$r ::= read(k)$	Read
$w ::= write(k,v) \mid delete(k)$	Update
$c ::= export(id) \mid import(id) \mid rollback$	Control

Figure 2: Transaction language syntax (subset).

3.1 Unrestricted operations

Clients access the storage by means of pre-declared transactions, similar to stored procedures in relational databases. These transactions, named armored-transactions, are written in a small transaction language, designed to meet three goals. First, it is a subset of popular general purpose programming languages, such as Java or C. This means that the syntax is familiar to developers, and the transaction specification could be easily embedded in larger programs. Second, the subset is large enough to support the basic operations required for expressive, multi-partition transactions. Third, the language makes explicit the points in the code where data may cross partition boundaries.

The language, whose syntax is in Figure 2, includes support for variable declarations and assignment; logical, relational, and arithmetic expressions; and basic control flow for branching and looping. Statements and expressions may be composed to express more complex logic.

At the heart of the transaction language are a set of built-in operations that are used to manipulate storage entries. These operations include *read* and *write* operations, as well as *delete*, to remove a key from the store.

Transaction-local variables persist for the duration of the transaction, and can be shared between partitions by using *export* and *import* control operations, which we explain in the next section. A transaction can request to abort its execution with a *rollback* operation.

Callinicos guarantees strict serializability for update transactions from all clients and read-only transactions submitted by correct clients. An update transaction contains at least one operation that modifies the state. Callinicos provides no guarantees for read-only transactions from faulty clients. More precisely, for every history H representing an execution 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.

3.2 Byzantine fault tolerance

Both clients and servers can present faulty behavior. To cope with faulty servers, each partition is fully replicated on a group of servers. Each server group uses atomic multicast and state machine replication [28, 39], implemented with PBFT [14], to ensure consistency. Therefore, although partitions can contain faulty servers, the partition as a whole follows the Callinicos protocol.

Faulty clients can attempt to disrupt the execution with a number of attacks, which include (a) leaving a transaction unfinished in one or more partitions, (b) forcing early termination of honest transactions (i.e., from a correct client), and (c) attempting denial-of-service attacks. We describe these attacks and the mechanisms to counter them in more detail in § 4.3.

3.3 Contention management

Callinicos implements on demand contention management. Every armored-transaction is multicast to each partition involved in the armored-transaction. The involved partitions are computed from the pre-declared armored-transaction, possibly reaching all partitions if the appropriate subset cannot be statically determined. Servers in a partition then try to acquire all the locks required by the armored-transaction. If all locks cannot be acquired, the servers request the ordering of the armored-transaction. This mechanism strives to avoid the overhead of ordering in the absence of contention, typical of distributed locking, and the penalty of aborts, typical of optimistic concurrency control.

4 Armored Transactions

Armored transactions implement a novel *multi-round* protocol, which generalizes the concept of mini-transactions by allowing data to flow across partitions in the context of the same transaction. The exchange of data between partitions in armored transactions is mediated by the clients that submit these transactions. Servers from each partition send the data to be transferred to the client at the end of a round, and the client forwards the data to the proper destination at the beginning of the next round. Figure 3 shows the complete execution for armored transactions. We explain the details in the following sections.

4.1 Transaction pre-processing

Before a client application can submit an armored-transaction for execution on the server, it first must transform the armored-transaction expressed in the Callinicos transaction language into a *transaction matrix*. A transaction matrix is a representation of the armored-transaction in which operations have been assigned to specific *partitions* and specific *rounds*. Each column in a transaction matrix represents a partition-specific sequence of rounds. Each row in the matrix represents the operations that will be executed in a single round. Intuitively, operations are mapped to a partition if they read or write data from that partition. Two operations, o_1 and o_2 are mapped to separate rounds if o_2 depends on the output of o_1, and they are executed on separate partitions.

Assuming a two-partition deployment, where key_a and key_b are on partitions 1 and 2, respectively, the transaction matrix for the swap operation shown in § 1 will be a 2×2 matrix (see Table 2). Both reads in the swap matrix are independent of any other operations and executed in the first round. Since the writes depend on the result of the reads, they are placed on the second round.

	Partition 1	Partition 2
Round 1	$a \leftarrow read(key_a)$ $export(a)$	$b \leftarrow read(key_b)$ $export(b)$
Round 2	$import(b)$ $write(key_a, b)$	$import(a)$ $write(key_b, a)$

Table 2: Transaction matrix for *Swap* command.

To build a transaction matrix, clients need to determine two pieces of information: (i) the data dependencies between operations, and (ii) the partitions where data is stored. Information about data dependencies can be learned by performing a static analysis of the armored-transaction logic to track data flow. Information about partition mappings can be learned by consulting the partition oracle. If the value of a key is known statically, then the partition oracle will tell the client where a key is stored. If the value of a key can only be determined at runtime, then the oracle will tell the client that no assertion about the data layout can be made, and instruct the

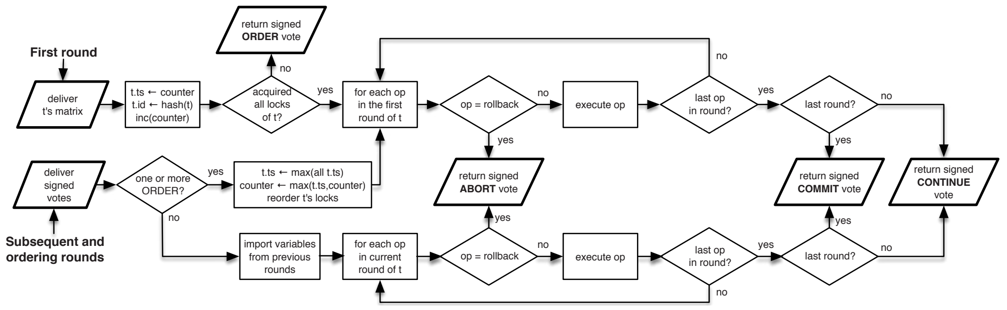


Figure 3: Armored-transaction execution on the servers in Callinicos.

client to execute a guarded version of the operation on all partitions (e.g., *if contains(k) then op(k)*).

Clients organize the operations in columns and rows as follows: (1) Operations that are confined to specific partitions are placed on their respective columns, or are executed on all partitions. (2) Operations that depend on the outcome of a previous operation executed on a different partition are placed on the row immediately after the row of the operation they depend on. (3) Independent operations are executed as early as possible.

After operators are assigned to rows and columns, each entry is augmented with *export* and *import* operations, which transport data across partitions at the end and beginning of each round, respectively. The details of the *export* and *import* operations, and the data transfer mechanism between rounds are presented in detail in § 4.2.

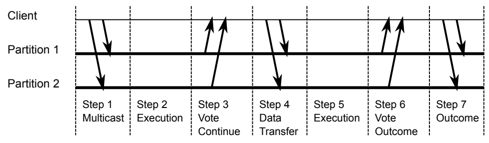
To reduce the implementation effort, our prototype relies on hand-written annotations that indicate partition and round number for each line of code in the transaction language specification. In a continuing development effort, we are working to automate the transformation.

4.2 Execution under normal conditions

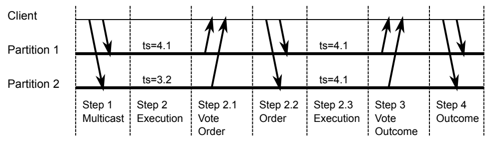
Once delivered for execution, an armored-transaction first transitions to a transient state (i.e., *ordering* or *pending*) and eventually reaches a final state (i.e., *committed* or *failed*). All state transitions are irreversible, irrevocable, and can be proved by signed certificates, which are generated as part of the state transition.

State transitions happen as the result of the *round executions*. Each round is composed of a request-reply exchange between a client and the servers. Requests from clients can start a new transaction, continue an ongoing transaction, or finalize a transaction. Replies from servers contain a signed vote with the round number, the outcome of the round, the current *timestamp* of the armored-transaction, and any applicable execution results. Servers implement a local and unique *counter*, used to assign timestamps to transactions, as described next. In the absence of failures and malicious behavior, an armored-transaction follows the sequence of steps described next. We discuss execution under failures in § 4.3.

Transaction submission. A client submits a new armored-transaction t by multicasting t's transaction matrix to each partition g involved in t. One multicast call is done per partition (see Figure 4, Step 1). Since the matrix is fully defined before it is multicast, every server knows how many rounds are needed to complete the transaction, which data should be returned in each round, and which data will be received in each round.



(a) with data transfer.



(b) with ordering

Figure 4: Execution of a two-round transaction.

The first round. Once server s delivers t's matrix, t becomes *delivered* at s and the value of s's counter becomes the timestamp ts of t. Each correct server s tries to acquire all the locks required by t (Figure 4, Step 2). If the lock acquisition fails (i.e., another transaction has a conflicting lock), s stops the execution of t and issues a signed vote for ORDER (Figure 4b, Step 2.1), meaning that s requires the next round of t to be an *ordering round* (i.e., t is in the *ordering* state at s). If s can acquire all locks needed by t, s will execute t's operations that belong to the current round. If s finds a rollback operation, it stops the execution of t, issues a signed vote for ABORT, meaning that it expects the current round to be the last and the next multicast from the client to provide a *termination certifi-*

cate. If s completes t's execution in the current round and the round is the last, s issues a signed vote for COMMIT (Figure 4a, Step 6). If the current round is not the last, s issues a signed vote for CONTINUE (Figure 4a, Step 3). In the case of a vote for CONTINUE, if the transaction contains any *export* operations in the current round, the respective variables are included in the signed vote. For any of these three votes t also becomes *pending*. In any case, a server's vote on the outcome of a round is final and cannot be changed once cast.

Lock management. A transaction can request *retrieve* and *update* locks on single keys and ranges of keys. Both single-key and key-range locks keep track of the acquisition order in a *locking queue*, i.e., a queue containing all the transactions that requested that specific lock, ordered by increasing timestamp. Retrieve locks can be shared; update locks are exclusive. A requested lock that conflicts with locks already in place will trigger an ordering round. Lock acquisition is performed sequentially, by a single thread, and at once for the complete transaction matrix, i.e., each server traverses its entire column and tries to acquire locks on all keys defined in all rounds. If a round performs retrieve or update operations on a key that is the result of a previous round, the server must perform a partition-wide lock. Callinicos's execution model avoids deadlocks since all the locks of a transaction are acquired atomically, possibly after ordering (defined next). If two conflicting multi-partition transactions are delivered in different orders in two different partitions, then each transaction will receive at least one vote to ORDER from one of the involved partitions, which will order both transactions.

Subsequent rounds. The execution of each following round starts with the client multicasting the vote certificate from the previous round (Figure 4a, Step 4). Each server then validates the signatures, tallies the votes, and proceeds according to the outcome of the certificate. The possible outcomes of the certificate are either: COMMIT, ABORT, ORDER, or CONTINUE.

If at least one vote is for COMMIT, or ABORT, then the transaction will terminate, as described below. If no partition votes for ORDER, t will transition to the *pending* state. Otherwise, t will go through an *ordering round* before transitioning to *pending* (Figure 4b, Step 2.2). The *pending* state indicates that a transaction has not yet been finalized (i.e., it has neither been committed nor aborted).

A vote for CONTINUE indicates that the previous round was not the last round of t. The transaction will remain in the *pending* state, and s executes the next round of t. The execution of the next round starts with the importing of variables exported in the previous round as indicated by *export* operations. These values are stored locally for the duration of the transaction, and are used in subsequent rounds. Once the execution of the round completes, the server issues a signed vote following the same rules as the first round. A server will not execute subsequent rounds until it has been presented with a vote certificate for the round it just executed. Furthermore, a server will not re-execute a round for which it has already issued a vote, unless it is an *ordering round*, as we explain next.

The ordering round. To order a transaction, the server examines the timestamps of all votes, selects the highest value, *maxts*, and sets t's timestamp to *maxts*. If s's counter is lower than *maxts*, s sets its counter to *maxts*. Then, t's locks are re-ordered and t is scheduled for re-execution (Figure 4b, Step 2.3). Because all (correct) servers in the partitions that contain t participate in the same protocol steps, t will have the same timestamp in all involved partitions. The re-ordering of t's locks ensures that for all locks ℓ acquired by t, there is no armored-transaction u with a timestamp lower than t ordered after t in ℓ's locking queue. We explain the re-ordering of locks below. Since a change of execution order may lead to a different outcome, t is re-executed when it becomes the transaction with the lowest timestamp that has no conflicts. The ordering round, if present, will always be the second round since lock acquisition is done in the first round. As a corollary, correct servers only vote for ORDER as the outcome of the first round. Furthermore, after an ordering round t is guaranteed to acquire all its locks since the criteria for execution requires all conflicting armored-transactions with a lower timestamp to finalize before t is executed. Notice that this does not undermine the irrevocability of the server's votes. Although the re-execution of a re-ordered armored-transaction creates a new vote, the new vote does not replace the vote from the first round; it becomes the vote of the ordering round.

Changing t's timestamp may (a) have no effect on ℓ's locking queue order, for each data item ℓ accessed by t; (b) change the order between non-conflicting armored-transactions (e.g., when re-ordering a retrieve lock with other retrieve locks); or (c) change the order between conflicting armored-transactions (e.g., an update lock is re-ordered after a series of retrieve locks). In case (a), ts was increased but no conflicting transaction is affected and thus t keeps its position in the locking queue. In cases (b) and (c), ts's increase resulted in a position change for t within ℓ's locking queue. In case (b), no transaction u with timestamp between t's old timestamp and ts conflicts with t and so, the position of t in ℓ's locking queue is adjusted but no further actions are necessary. For case (c), the locking state of each conflicting transaction u has to be updated to assess whether u is eligible for re-execution (i.e., if u is the transaction with the lowest timestamp and has no conflicts).

Transaction termination. If the incoming vote certificate contains at least one partition vote for ABORT or COMMIT votes from all partitions, s treats the vote certificate as a *termination certificate*, i.e., a certificate that is used to determine the *outcome* of t (Figure 4a, Step 7). If the outcome is ABORT, s rolls back t by discarding any update buffers; if COMMIT, s applies t's updates. In either case, t is no longer *pending* and its locks are released.

4.3 Execution with faulty clients

Faulty clients can attempt to disrupt the protocol by violating *safety* or *liveness* guarantees. The armored-transaction protocol builds on our prior work on Augustus [35] to ensure that safety cannot be violated. Callinicos does not provide absolute liveness guarantees. However, Callinicos does provide protection against some specific attacks. Below, we describe three in particular: (a) leaving a transaction unfinished in one or more partitions, which can also happen when a client fails by crashing; (b) forcing early termination of honest transactions (i.e., from a correct client); and (c) attempting denial-of-service attacks.

Unfinished transactions. A faulty client may leave a transaction unfinished by not executing all transaction rounds. We address this scenario by relying on subsequent correct clients to complete pending transactions left unfinished. If a transaction t conflicts with a pending transaction u in server $s \in g$, s will vote for ordering t. In the ORDER vote sent by s to the client, s includes u's operations. When the client receives an ORDER vote from $f_g + 1$ replicas in g, it forwards the ordering vote certificate for t and starts the termination of u by multicasting u's operations to every partition h involved in u. Clients do not have to immediately start the recovery of u, in particular when u is a multi-round transaction.

Once t is ordered, the client has a guarantee that t will eventually be executed. The amount of time that the client should wait before trying to recover u can be arbitrarily defined by the application. 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). If u is a single-round transaction, then the client will gather a vote certificate to finalize u. If u is a multi-round transaction, then the client will gather a vote certificate for the first execution round. In any case, eventually the client will gather enough votes to continue or finalize the execution of u, following the same steps as the failure-free cases.

Forced early termination of honest transactions. Faulty clients cannot force an erroneous or early termination of a honest transaction t. The atomic multicast protocol ensures that faulty clients cannot tamper with each others' transactions prior to delivery. Since we assume that faulty processes cannot subvert the cryptographic primitives, it is impossible for faulty clients to forge a transaction that will match the id of t once t is delivered. Furthermore, it is impossible for faulty clients to forge the vote certificates. Thus, once t is delivered to at least one partition, that partition will be able to enlist help from correct clients to disseminate t to the other partitions through the mechanism used to address unfinished transactions, and complete the execution of t.

Denial-of-service attacks. Faulty clients can try to perform denial-of-service attacks by submitting either: (a) transactions with many update operations, (b) multiple transactions concurrently, or (c) transactions with a very large number of rounds. Although we do not currently implement them in our prototype, a number of measures can be taken to mitigate such attacks, such as limiting the number of operations in a transaction, restricting the number of simultaneous pending transactions originating from a single client (e.g., [29]), or limiting the number of rounds in a transaction. These attacks, however, cannot force honest transactions to abort.

4.4 Correctness

In this section, we argue that for all executions H produced by Callinicos 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, recovering an unfinished transactions 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 that access partitions in PS (partition set) and PS', respectively.

First, assume that PS and PS' intersect and $p \in PS \cap PS'$. There are two possibilities: (i) either the operations requested by T and T' do not conflict, or (ii) at least one operation in each transaction conflict, and the transactions need to be ordered. In (i), property (a) is trivially ensured since neither transaction reads from the other, otherwise they would conflict. To ensure (b), T and T' appear in H_s following their termination order, if they are not concurrent. If T and T' are concurrent, then their order in H_s does not matter. In (ii), from the algorithm (ii.a) T commits in every p before T' is executed at p, or (ii.b) T' commits in every p before T is executed at p, or (ii.c) T is executed first in a subset p_T of p and T' is executed first in the remaining (and complementary) subset $p_{T'}$ of p. For cases (ii.a) and (ii.b), without lack of generality, we 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 will commit regardless of the outcome of T'. Property (b) holds because it impossible for T' to execute before T.

For case (ii.c), the vote certificate for T will contain COMMIT votes from p_T and ORDER votes from $p_{T'}$. Each of these votes contains a unique timestamp for T. The presence of an ORDER vote from a partition in PS forces all of PS to update the timestamp of T to the largest timestamp observed in the vote certificate (i.e., the final timestamp), and adjust the execution order of T according to this timestamp. If the final timestamp of T is smaller than the final timestamp of T', then T will be executed before T', and thus T precedes T' in H_s.

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 an unfinished transaction will extend its lifetime, but will not change the argument above.

Case 3. T is a single partition transaction that accesses partition P and T' is a multi-partition transaction that accesses 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' will precede T based on its final timestamp. If T' does not yet have a final timestamp, then T has to wait. If the final timestamp of T' remains smaller than the timestamp of T, 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', as T has to wait for T'. Otherwise, T' will be such that T is executed before T', and thus T precedes T' in H_s as explained above.

5 Evaluation

Callinicos is designed to address three critical limitations of mini-transactions related to reliability, expressivity, and performance. These limitations manifest themselves in workloads that are *inherently unscalable*. Prior work on storage systems designed for cross-partition data-exchange or high contention workloads typically relax consistency requirements in order to meet performance demands (e.g., [48]). This observation is also reflected in the design of common storage system benchmarks. For example, widely used benchmarks like TPC-C [45] avoid queries that result in “hotspots”. For this reason, we evaluate Callinicos using a set of micro-benchmarks. These micro-benchmarks are both inspired by real-world applications and illustrate the benefits of Callinicos on the types of workloads that cause most systems to perform poorly. Overall, our experiments demonstrate that Callinicos allows developers to easily build distributed applications with performance that scales with the number of partitions for high contention workloads.

Our prototype is implemented in Java 7. It includes an atomic multicast implementation based on PBFT [14] and the transaction processing engine. All source code is publicly available under an open source license.[1] In the experiments, each client is a thread performing synchronous calls to the partitions sequentially, without think time. Each partition contained four servers.

We ran all the tests on a cluster with the following configuration: (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) an HP ProCurve Switch 2910al-48G gigabit network switch. The single-hop latency between two machines connected to different switches is 0.17 ms for a 1KB packet. The nodes ran CentOS Linux 6.5 64-bit with kernel 2.6.32. We used the Sun Java SE Runtime 1.7.0_40 with the 64-Bit Server VM (build 24.0-b56).

5.1 Kassia distributed message queue

The first set of experiments measures the scalability of Callinicos, and evaluates the benefits of cross-partition communication. As a baseline, we compare the results against a mini-transaction implementation of the same queries. Although there are many examples of distributed data structures, including B-trees [1] and Heaps [11], we chose to focus on distributed queues because (i) the implementation of their basic operations are easy to explain, (ii) they are widely used by a number of real-world systems, including those at Yahoo, Twitter, Netflix, and LinkedIn [37], and (iii) they exhibit a natural point of contention, since every producer must read the location of the queue head when executing a *push* operation.

Our distributed message queue service, named Kassia, is inspired by Apache Kafka [5]. Both Kassia and Kafka implement a partitioned and replicated commit log service, which can be used to support the functionality of a messaging system. In both Kassia and Kafka, *producers*

[1] https://github.com/usi-systems/callinicos

generate new messages and publish them to *queues*, while *consumers* subscribe to queues. In both systems, queues can be assigned to different partitions. Kassia offers the same abstractions as Kafka, but with two crucial differences. First, while Kafka is only able to ensure a total ordering of messages on a single partition [6], Kassia (i.e., Callinicos) ensures a total ordering of messages across multiple partitions. Thus, Kassia, can distribute the load of message production and consumption by increasing the number of partitions. Second, while Kafka provides no reliability guarantees, Kassia tolerates Byzantine failures.

A message in Kassia is an entry of type (*msg_id*, *msg_content*), where *msg_id* is a key in the key-value store. A message queue is a sequence of pointers to *msg_id*'s. A designated storage area (i.e., the *head index*) is used to maintain an index that points to the first position in the queue. A second designated storage area (i.e., the *tail index*) is used to maintain an index that points to the last position in the queue.

To publish a message, a producer must perform the following operations: (i) read from the tail index to learn the location of the tail of the queue; (ii) increment the tail of the queue to the next location; (iii) write the message to the new tail location; and (iv) update the tail of the queue in the tail index.

Consumers in Kassia never remove messages from the queue. Instead they perform a sequential scan of all messages from the head to the tail. Thus, consumers perform the following operations: (i) read from the tail index to learn the location of the tail of the queue; (ii) read from the head index to learn the location of the head of the queue; (iii) issue a read range request to read all messages in the queue.

Thus, we see that producer operations exhibit high-contention, since all producer transactions include a write to the same location in the store. In contrast, consumer operations exhibit low-contention, read-only workloads.

We implemented the Kassia producer and consumer transactions using both armored-transactions and mini-transaction versions. Like the *swap* example from § 1, the mini-transaction version needed to be split into two separate transactions. The first transaction reads the tail index, while the second needs to use a compare operation to ensure that the value hasn't changed between the execution of the first and second transactions.

Scalability of Callinicos transactions. The first experiment evaluates how armored-transactions scale. We measured the maximum throughput for producer transactions as we increased the number of partitions. For this experiment, a separate queue was created in each partition and there were approximately six producers per queue. Thus, the workload itself is scalable. In other words, if all producers wrote to a single queue, we would not expect to see armored-transactions scale.

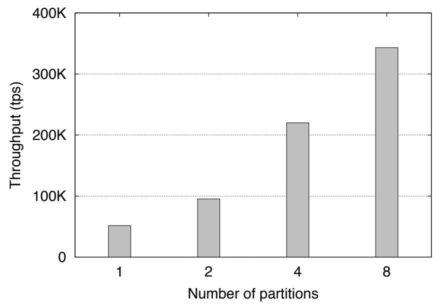


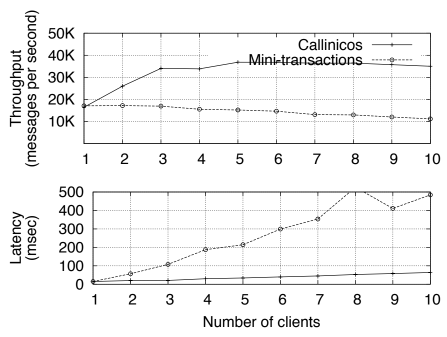
Figure 5: Maximum throughput for the producer (in transactions per second) with increasing number of partitions.

As shown in Figure 5, Callinicos scales nearly linearly with the number of partitions, achieving peak throughput of 60k, 116k, 220k and 350k messages per second with 1, 2, 4 and 8 partitions, respectively. In this experiment, producers submit batches with 200 messages of 200 bytes.

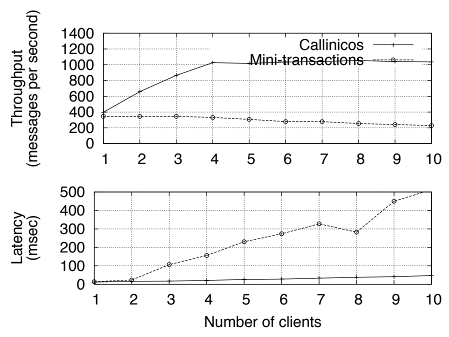
Armored-transactions vs. mini-transactions. The next experiment considers the question of how armored-transactions perform compared to mini-transactions for *high-contention* and *low-contention* workloads. For high-contention workloads, an initially empty queue was repeatedly written to by increasing number of producers. For low-contention workloads, an initially full queue was repeatedly read by increasing numbers of consumers. Recall that consumers do not remove messages from the queue; each consumer operation reads all the messages in the queue. For both workloads, we measured the throughput and latency as we increased load.

For this experiment, there was a single queue distributed across 4 partitions. Thus, in contrast to the scalability measurements above, the workload is inherently unscalable. As a further parameter to the experiment, we varied the number and size of messages sent by the clients. Recall that each client executes a single thread that repeatedly sends batches of synchronous messages. In the first configuration, clients send 200 messages of 200 bytes in each batch. In the second configuration, they send 4 10-KB messages in each batch.

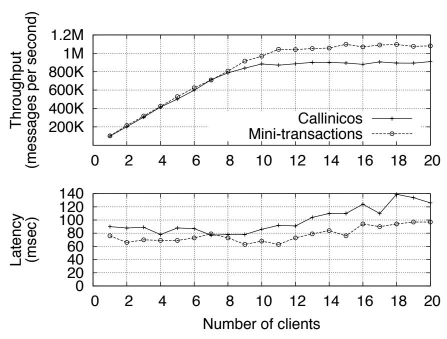
Figures 6 (a) and (b) show the producer results. With 200-byte messages, Callinicos outperforms mini-transactions at peak throughput by a factor of 2.1. Moreover, due to the high number of aborts experienced by mini-transactions, producers need to resubmit their requests, which increases latency. With 10k-byte messages, Callinicos outperforms mini-transactions by a factor of 2.5. Both systems present similar latency up to 2 clients, but the latency of mini-transactions increases quickly with larger number of clients. In both configurations, we can



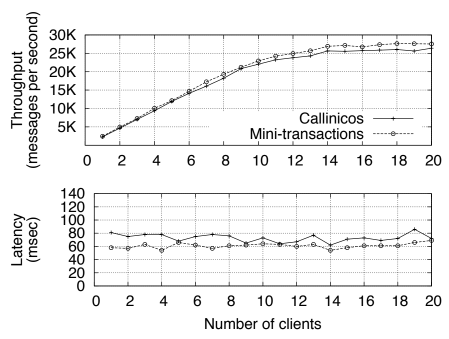
(a) 200-byte messages (producer)



(b) 10k-byte messages (producer)



(c) 200-byte messages (consumer)



(d) 10k-byte messages (consumer)

Figure 6: Compared to mini-transactions, Kassia producers demonstrate higher throughput and lower latency compared to mini-transactions for contention-heavy workload. Kassia consumers show no overhead for contention-light workload.

also observe that the two system behave differently once they reach their point of saturation: armored-transactions experience quite stable performance, while in the case of mini-transactions performance drops.

Figures 6 (c) and (d) show the consumer results. In the absence of contention, mini-transactions perform slightly better than Callinicos armored-transactions with small messages; with larger messages the difference between the two systems is negligible. In both cases, mini-transactions consistently display lower latency than armored-transactions, although as for throughput, the difference is more noticeable with small messages. Since the consumer workload is read-only, there are no aborts and both systems sustain throughput at high load.

Our experiments show that Callinicos provides better throughput and latency than mini-transactions for high-contention workloads. For low-contention workloads, Callinicos adds no additional overhead.

5.2 Buzzer distributed graph store

As a second application, we implemented a Twitter clone named Buzzer, which is backed by a distributed graph store. Most social networking applications exhibit only eventually-consistent semantics [48]. As a result, users have become accustomed to odd or anomalous behavior (e.g., *one friend sees a post, and another doesn't*). In contrast, Buzzer not only provides strict serializability, but also tolerates Byzantine failures. Moreover, graph stores offer a useful point-of-comparison because: (i) they demonstrate a different design from queues where contention is less pronounced (i.e., all writers don't update the same memory location), and (ii) graphs have become increasingly popular for applications in telecommunications, transportation, and social media [42].

In Buzzer, there are operations to *post* a message, *follow* a user (friends), and to retrieve one's *timeline* (i.e., an aggregate of the friends' posts). *Posts* are single-round, single-partition update transactions. *Follows* are single-round, multi-partition update transactions. *Timelines* are multi-round, multi-partition read-only transactions.

The level of contention in a graph store depends on the workload and the structure of the graph on which the workload is run. For these experiments, we used a single workload that was composed of 85% *timeline*, 7.5% *follow*, and 7.5% *post* operations, and varied the contention by altering the connectivity of the graph. For typical graph queries that update data as they traverse the graph, dense, highly connected graphs exhibit high contention, while sparse graphs exhibit low contention. We opted for a small social network with 10,000 users. Then, using statistics from Twitter [46], we inferred that the "friending" behavior of Twitter users approximately follows a Zipf distribution with size 2,500 and skew of 1.25. We built our social network graph by sampling that Zipf distribution. We call this network *high-contention*. To contrast with this highly connected network, we created another network using a Zipf distribution with size 25 and skew 1.25. We call this network *low-contention*.

In these experiments, we explored the question of *how Callinicos' conflict resolution compares to optimistic concurrency control*. The experiments measure the maximum throughput for the workload described above.

While partitioning the queue data was relatively straightforward, ensuring a good partitioning of the social network data is somewhat more complex. We distributed relationships such that each user has a 50% probability of having her friends' data on the same partition where her data is. If not on the same partition, all friends' data is placed on two partitions, the user's partition and another partition chosen randomly. For example, for the four-partition scenario all data accessed by a user has a 50% chance of being fully contained in a single partition and a 50% chance of being spread across two different partitions (16.67% for each combination). We enforced this restriction to assess the scalability of the system as the number of partitions increase.

The results, seen in Figure 7, show that Callinicos' conflict management adds little overhead for low-contention workloads. However, for high-contention workloads, Callinicos' demonstrates significantly better throughput. In other words, although ordering conflicting transactions adds a slight overhead, it avoids aborts which are even more detrimental to performance. Moreover, these experiments show that for both high-contention and low-contention workloads, the throughput of Callinicos scales with the number of available partitions.

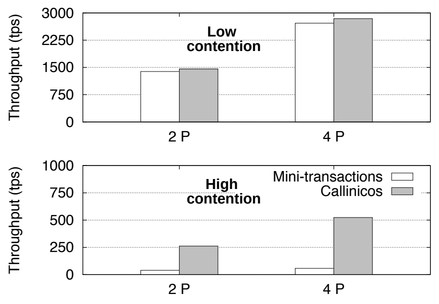


Figure 7: Buzzer's maximum throughput under low- and high-contention for 2 and 4 partitions (2 P and 4 P).

6 Related work

Many storage systems have been designed and implemented. In this section, we compare Callinicos to many of these systems from a few different perspectives.

Distributed storage systems. Database systems that implement some notion of strong consistency (e.g., serializability, snapshot isolation) traditionally use two-phase locking (2PL), optimistic concurrency control (OCC), or a variation of the two to coordinate the execution of transactions. Examples of distributed database systems based on 2PL are Gamma [20], Bubba [13], and R* [32]. Spanner [17] uses 2PL for update transactions and a timestamp-based protocol for read-only transactions. Examples of recent systems based on OCC are H-Store [25] and VoltDB [49]. MDCC [27] and GeoDUR [40, 41] use OCC for geo-replicated storage. Percolator implements snapshot isolation using OCC [12]. In a seminal paper, Gray et al. [24] have shown that two-phase locking and optimistic concurrency control are exposed to a prohibitively high number of deadlocks and aborts when used to handle replication. Fundamentally, the problem stems from the fact that requests are not ordered among replicas, a limitation that is addressed by Callinicos.

Storage systems with limited transactions. Several distributed storage systems improve performance by supporting limited types of distributed transactions. For example, MegaStore [7] only provides serializable transactions within a data partition. Other systems, such as Granola [18], Calvin [44] and Sinfonia [2] propose concurrency control protocols for transactions with read/write keys that are known a priori. Moreover, many cloud-oriented storage systems have abandoned transactional properties to improve performance. For example, Apache Cassandra [3], Apache CouchDB [4], MongoDB [33], and Amazon Dynamo [19] offer no transaction support. When using such systems, applications must handle contention explicitly, by preventing conflicts from happening or by allowing weaker consistency (e.g., eventual consistency). NoSQL scalable storage systems that offer transactions are usually limited to single-row updates (e.g., [15]) or single-round transactions (e.g., [2, 18]), and employ optimistic concurrency control (e.g., [2, 7, 35]).

Storage systems with support for contention. A few systems have been proposed that can handle contention, although none of these systems tolerate Byzantine failures. Calvin [44] deals with contention by preemptively ordering all transactions. It adds a sequencing layer on top of any partitioned CRUD storage system and enables full ACID transactions. The sequencing layer operates in rounds. Each round lasts 10 ms, which is used to batch incoming requests. Once a sequencer finishes gathering requests for a given round, it exchanges messages with other sequencers to merge their outputs. The sequencing layer provides ordering without requiring locks, and supports transactional execution without a commit round. Calvin, however, does not support multi-round execution.

Rococo [34] uses a dependency graph to order transactions. Transactions are broken into pieces that can exchange data. Pieces are either immediate or deferrable. An offline checker analyzes all transactions in the system and decides which can be reordered based on their pieces. Once executed, an immediate piece cannot be reordered, i.e., transactions with conflicting immediate pieces cannot be reordered. A central coordinator distributes pieces for execution, forwards intermediary results, and collects dependency information. At commit, each server uses the dependency graph information to reorder deferrable pieces. Callinicos can reorder all types of transactions and does it using a simpler algorithm based on timestamps.

H-Store [43] promotes the idea of favoring single-partition transactions executed in a single thread without any contention management. This approach ensures optimal performance under no contention. When the number of aborts increases (i.e., under heavier contention), H-Store first tries to spread conflicting transactions by introducing waits, and then switches to a strategy that keeps track of read and write sets to try to reduce aborts. Callinicos keeps track of the read and write sets by default, and orders and enqueues conflicting transactions instead of aborting them. Multi-round execution in H-Store depends on a centralized coordinator, which is responsible for breaking transactions into *subplans*, submitting these subplans for execution, and executing application code to determine how to continue the transaction.

A speculative execution model on top of H-Store is proposed in [25], where three methods of dealing with contention are compared. The first method, *blocking*, simply queues transactions regardless of conflict. The second method, *locking*, acquires read and write locks, and suspends conflicting transactions. If a deadlock is detected, the transaction is aborted. The third method is *speculative execution*. Conflicting multi-partition transactions are executed in each partition as if there were no contention, and if they abort due to contention their rollback procedure requires first rolling back all subsequent transactions and then re-executing them. Multi-round execution follows the model proposed by H-Store. Experimental evaluation of the speculative model shows that for workloads that incur more than 50% of multi-partition transactions, the throughput of the speculative approach drops below the locking approach. In a multi-round transaction benchmark, the throughput of the speculative model dropped to the level of the blocking approach.

Granola [18] uses timestamps to order transactions. The timestamps are used in two different types of transactions: *independent* and *coordinated*. For independent transactions, replicas exchange proposed timestamps, select the highest proposal, and execute the transaction at the assigned timestamp. Coordinated transactions also exchange timestamp proposals to order transactions, but they abort if any replica votes ABORT or detects a conflict.

Byzantine fault-tolerant storage systems. Some storage systems have been proposed that can tolerate Byzantine failures. MITRA [30, 31] and Byzantium [22] are middleware-based systems for off-the-shelf database replication. While MITRA implements serializability, Byzantium provides snapshot isolation. BFT-DUR [36] and Augustus [35] are transactional key-value stores. BFT-DUR is an instance of deferred update replication and can support arbitrary transactions. Augustus provides an interface similar to Sinfonia, with transactions with operations defined a priori. All these systems rely on optimistic concurrency control to ensure strong consistency. Thus, in the case of contention they are exposed to many aborts. HRDB [47] provides BFT database replication by relying on a trusted node to coordinate the replicas. Although HRDB provides good performance, the coordinator is a single point of failure. DepSky [9] provides confidentiality for cloud environments. It offers users a key-value interface without the abstraction of transactions.

7 Conclusion

Callinicos introduces a generalized version of the single-round mini-transaction model. Multi-round armored-transactions are modeled using a transaction matrix, in which rows indicate the rounds, columns indicate the partitions, and each cell contains the commands that a partition will execute during a round. Callinicos allows for cross-partition communication, and, uses a timestamp-based approach to reorder and ensure the execution of conflicting transactions. Using Callinicos, we've implemented Kassia, a distributed BFT message queue, and Buzzer, a social network backed by a distributed graph store. that scales with the number of partitions. Overall, our experiments show that Callinicos offers scalable performance for high-contention workloads, and that the design allows users to implement robust, performant distributed data structures.

Acknowledgements

We wish to thank Benjamin Reed, our shepherd, and the anonymous reviewers for the constructive comments. This work was supported in part by Microsoft Research through its PhD Scholarship Programme and the Hasler Foundation.

References

[1] Aguilera, M. K., et al. A practical scalable distributed b-tree. *Proc. VLDB Endow. 1*, 1 (2008), 598–609.

[2] Aguilera, M. K., et al. Sinfonia: A new paradigm for building scalable distributed systems. *TOCS 27*, 3 (2009), 5:1–5:48.

[3] Apache Cassandra. `http://cassandra.apache.org/`.

[4] Apache CouchDB. `http://couchdb.apache.org/`.

[5] Apache Kafka. `http://kafka.apache.org/`.

[6] Kafka 0.9.0 Documentation. `https://kafka.apache.org/documentation.html`.

[7] Baker, J., et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011).

[8] Bartlett, W., and Spainhower, L. Commercial fault tolerance: A tale of two systems. *IEEE TDSC 1*, 1 (Jan. 2004), 87–96.

[9] Bessani, A., et al. Depsky: Dependable and secure storage in a cloud-of-clouds. In *EuroSys* (2011).

[10] Bessani, A., et al. State machine replication for the masses with bft-smart. In *DSN* (2014).

[11] Bhagwan, R., et al. Cone: A distributed heap-based approach to resource selection. Tech. rep., UCSD, December 2004.

[12] Bhatotia, P., et al. Large-scale incremental data processing with change propagation. In *HotCloud* (2011).

[13] Boral, H., et al. Prototyping bubba, a highly parallel database system. *IEEE TKDE 2*, 1 (Mar. 1990), 4–24.

[14] Castro, M., and Liskov, B. Practical byzantine fault tolerance and proactive recovery. *TOCS 20*, 4 (November 2002), 398–461.

[15] Chang, F., et al. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[16] Clement, A., et al. BFT: The time is now. In *LADIS* (2008), pp. 1–4.

[17] Corbett, J. C., et al. Spanner: Google's globally distributed database. *ACM TOCS 31*, 3 (Aug. 2013), 8:1–8:22.

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

[19] DeCandia, G., et al. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[20] Dewitt, D. J., et al. The gamma database machine project. *IEEE TKDE 2*, 1 (Mar. 1990), 44–62.

[21] Dwork, C., et al. Consensus in the presence of partial synchrony. *JACM 35*, 2 (April 1988), 288–323.

[22] Garcia, R., et al. Efficient middleware for byzantine fault tolerant database replication. In *EuroSys* (2011).

[23] Gray, J. A census of tandem system availability between 1985 and 1990. *IEEE TOR 39*, 4 (1990), 409–418.

[24] Gray, J., et al. The dangers of replication and a solution. *SIGMOD Rec. 25*, 2 (June 1996), 173–182.

[25] Jones, E. P., et al. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD* (2010).

[26] Kotla, R., et al. Zyzzyva: Speculative byzantine fault tolerance. *ACM TOCS 27*, 4 (2010), 1–39.

[27] Kraska, T., et al. Mdcc: Multi-data center consistency. In *Eurosys* (2013).

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

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

[30] Luiz, A. F., et al. Byzantine fault-tolerant transaction processing for replicated databases. In *NCA* (2011).

[31] Luiz, A. F., et al. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec. 43*, 1 (May 2014), 32–38.

[32] Mohan, C., et al. Transaction management in the R* distributed database management system. *ACM TODS 11*, 4 (Dec. 1986), 378–396.

[33] MongoDB. `http://www.mongodb.org/`.

[34] Mu, S., et al. Extracting more concurrency from distributed transactions. In *OSDI* (2014).

[35] Padilha, R., and Pedone, F. Augustus: Scalable and robust storage for cloud applications. In *Eurosys* (2013).

[36] Pedone, F., and Schiper, N. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society 18*, 1 (2012), 3–18.

[37] Powered By Apache Kafka. `https://cwiki.apache.org/confluence/display/KAFKA/Powered+By`.

[38] Prabhakaran, V., et al. Iron file systems. In *SOSP* (2005), pp. 206–220.

[39] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR 22*, 4 (1990), 299–319.

[40] Sciascia, D., et al. Scalable deferred update replication. In *DSN* (2012).

[41] Sciascia, D., and Pedone, F. Geo-replicated storage with scalable deferred update replication. In *DSN* (2013).

[42] Soulé, R., and Gedik, B. Railwaydb: adaptive storage of interaction graphs. *The VLDB Journal* (2015), 1–19.

[43] Stonebraker, M., et al. The end of an architectural era (it's time for a complete rewrite). In *SIGMOD* (2007).

[44] Thomson, A., et al. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012).

[45] TPC-C. `http://www.tpc.org/tpcc/`.

[46] Twitter Statistics. `http://www.beevolve.com/twitter-statistics/`, Oct. 2012.

[47] Vandiver, B., et al. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *SOSP* (2007).

[48] Venkataramani, V., et al. Tao: How facebook serves the social graph. In *SIGMOD* (2012).

[49] VoltDB. `http://www.voltdb.com/`.

[50] Yang, J., et al. Explode: A lightweight, general system for finding serious storage system errors. In *OSDI* (2006).