CrossStitch: An Efficient Transaction Processing Framework for Geo-Distributed Systems

Sharon Choy, Bernard Wong, Xu Cui, Xiaoyi Liu
Cheriton School of Computer Science, University of Waterloo
s2choy, bernard, xcui, x298liu@uwaterloo.ca

Abstract

Current transaction systems for geo-distributed datastores either have high transaction processing latencies or are unable to support general transactions with dependent operations. In this paper, we introduce CrossStitch, an efficient transaction processing framework that reduces latency by restructuring each transaction into a chain of state transitions, where each state consists of a key operation and computation. Transaction states are processed sequentially, and the transaction code and data is sent directly to the next hop in the chain. CrossStitch transactions can be organized such that all states in a location are processed before transitioning to a state in a different location. This allows CrossStitch to significantly reduce the number of inter-location crossings compared to transaction systems that retrieve remote data to a single location for processing. To provide transactional properties while preserving the chain communication pattern, CrossStitch introduces a pipelined commit protocol that executes in parallel with the transaction and does not require any centralized coordination. Our evaluation results show that CrossStitch can reduce the latency of geo-distributed transactions when compared to a traditional 2PC-based distributed transaction system. We demonstrate that CrossStitch can reduce the number of round trips by more than half for TPC-C-like transactions.

1. Introduction

Data is being generated at a rapidly increasing rate around the world. Many believe that effectively using and managing this data is one of the next great computing challenges [1]. This challenge is especially difficult for interactive applications with strict latency requirements. Providing low write latency for these applications requires a storage system that stores data near the data source. However, providing low read latency requires geo-replicating the data to reduce the number of high-latency network round trips. These opposing requirements have led to significant work on trading off consistency for lower latency [12]. By reducing consistency guarantees, data updates can be performed locally and then disseminated asynchronously to other geographic locations. Unfortunately, not all applications can accept weak consistency guarantees. This includes any application that uses transactions as transactional requirements demand a strongly consistent storage system.

Although these interactive applications may have strict latency requirements, many of them have sufficient latency budgets to allow for a small number of remote read or write operations. For simple transactions where all data operations can be executed in parallel, an application can use Sinfonia [7] to combine data operations with the commit protocol to complete the transaction in only two network round-trips. Alternatively, for predefined transactions that are conducive to transaction chopping [21], an application can use Lynx [21] to transform a transaction into a chain of smaller transactions. Through static analysis, Lynx can ensure that if the first hop of the chain commits, all of the remaining hops will eventually commit. This can reduce the number of network round trips before a transaction returns control to the application. However, Lynx does not reduce the total transaction completion time or preserve transaction ordering, which means that transaction T_2 may be serialized before T_1 even though the client waited until T_1 returned before submitting T_2.

In this paper, we introduce CrossStitch, an efficient transaction processing framework for geo-distributed datastores. CrossStitch supports general transactions without any operational restrictions. Instead of specifying a transaction as a sequential set of computation and data operations between a pair of `begin` and `commit` statements, CrossStitch transactions are modelled as a series of state transitions where each state consists of a single key operation and arbitrary computation. A CrossStitch transaction is passed, on each state transition, to the storage server responsible for the

Technical Report - CS-2015-08, June 26, 2015, University of Waterloo, Waterloo, Ontario.

next state's key. Upon receiving a transaction, the storage server performs the state's key operation and computation, and sends the transaction to the next server responsible for the next state transition. CrossStitch's transaction processing model allows for fewer network round trips on the critical path than traditional transaction processing systems that execute the entire transaction on a single client using remote read and write operations as needed or systems that combine code-shipping with a tree-based commit protocol [17].

Moreover, this model enables CrossStitch transactions to have a communication pattern similar to RPC Chains [19], in which back-and-forth traffic between the client and storage servers at different geographical locations are replaced with a single chain of state transitions across storage servers. By providing the transaction with the key's location information, the CrossStitch framework offers the transaction writer the tools to construct a chain in a way that minimizes the number of inter-location crossings.

The key challenge to providing a chain communication pattern for transactions is in eliminating centralized transaction coordination while still providing a fault-tolerant mechanism to atomically commit or abort each transaction. Using traditional two-phase commit (2PC) with CrossStitch transactions would require forwarding each transaction to the coordinator on each hop in the chain. This is necessary to accurately track storage server membership for a transaction in the event of a server or network failure. Therefore, 2PC or any coordinator-based atomic commit protocol would create back-and-forth traffic between the coordinator and the storage servers and be unable to preserve CrossStitch's chain communication pattern.

CrossStitch addresses this problem by introducing a pipelined commit (PLC) protocol that is loosely based on linear 2PC [2], but can execute in parallel with a CrossStitch transaction such that the transaction can partially commit on each state transition. Much like linear 2PC, PLC only requires that the transaction manager at each storage server communicate with the transaction managers at the next and previous hop in the chain. However, PLC does not need to wait until the transaction is ready to commit or abort before initiating the commit protocol. This is possible because PLC exchanges an additional message between adjacent transaction managers to discover transaction managers further down the chain before indicating that it is ready to commit. This additional information enables PLC to atomically commit or abort a transaction even in the event of up to k server or network failures, while requiring at most k additional network round-trips on the critical path.

We evaluate CrossStitch using both synthetic workloads and TPC-C transactions on a working prototype. Our results show that, by rewriting the transactions into a series of state transitions, CrossStitch can reduce the number of network round-trips between different geographical locations by more than half compared to a traditional 2PC-based distributed transaction system.

Overall, our work makes three contributions:

- We present a novel approach of restructuring general transactions into a series of state transitions where each state consists of a key operation and computation.
- We introduce a pipelined commit protocol that, when combined with state transition-based transactions, can preserve CrossStitch's chain communication pattern.
- We evaluate the performance of a working CrossStitch prototype and show that it requires less than half the number of inter-location crossings than a traditional 2PC-based distributed transaction system for TPC-C-like transactions.

2. Background and Related Work

Many distributed datastores [9, 12, 15] trade off features such as transactional support in favour of obtaining high availability and low response time for their clients. Nonetheless, reliable transaction processing remains critical for applications that run on these geo-distributed datastores. Examples of such applications include social networking, collaborative editing, and financial management. Transactions, which provide ACID (atomicity, consistency, isolation and durability) properties, are necessary to ensure the integrity of the underlying data.

In order to provide distributed transactional support, traditional transaction processing systems, particularly those that follow the XOpen model, exchange a significant number of messages between the client, the transaction's coordinator, and participating servers. Consequently, this may cause the completion time of a transaction to increase, particularly in the case where the client, coordinator and participating servers are not in the same geographical location

A popular atomic commit protocol that is frequently used in distributed storage systems is the two-phase commit protocol. This protocol requires a significant number of messages between the coordinator and the participating servers. Additional optimizations, such as linear 2PC, can reduce the number of messages. In linear 2PC, vote requests are not sent concurrently; instead, the coordinator and neighbouring participants vote in a linear order. However, by still relying on a coordinator, linear 2PC does not reduce the number of cross-datacenter roundtrips compared to 2PC. In addition to the 2PC and linear 2PC, Mohan et al. [17] present the R* commit protocol that organizes the commit messages and the coordinator and participants into a tree. The R* commit protocol is an extension of 2PC which reduces inter-datacenter message traffic and log writes.

2.1 Geo-Distributed Storage Systems

Many existing geo-distributed storage systems are designed so that they may serve their clients locally in order to re-

duce latency. Most of these systems offer a simple key-value interface to access and store data, while some offer a multi-dimensional map structure [8, 10].

Geo-distributed storage systems can reduce end-user latency by using local replicas to serve their clients. This comes at a cost of having a higher write latency to ensure that updates are propagated to all replicas. To improve performance, these systems may also use a weaker consistency model. For example, Dynamo [12] provides eventual consistency, which in turn allows updates to be propagated to servers asynchronously, but requires that end-users reconcile diverging versions of data. Some geo-distributed storage systems, such as Yahoo's PNUTs [9] and Cassandra [15], can provide stronger consistency guarantees. However, PNUTs does not offer serializable transactions, which makes it unsuitable for some applications. Cassandra does provide support for lightweight transactions, but lightweight transactions are restricted to keys on a single partition [4]. This is not suitable for use in geo-distributed storage systems where transactions may span multiple partitions in different datacenters.

2.2 Transaction Processing Frameworks

More recent geo-distributed storage systems provide transactional support. However, they may not be appropriate for a general workload or for the public cloud. For example, Megastore [8] partitions its data, replicates partitions separately, and provides serializable transactions within a partition. The write latency for Megastore ranges from 100-400 ms, which may be too high for latency-sensitive applications. Google also developed Spanner [10], which relies on knowledge of clock uncertainty and time synchronization to provide external consistency. However, Spanner relies on centralized transaction managers to coordinate transactions that span different Paxos [16] groups, which can lead to a significant number of cross-datacenter roundtrips when the Paxos groups are not in the same geographical location. Moreover, Spanner is designed for long-lived transactions; thus, its design may not be suitable for many short-lived geo-distributed transactions that execute in the cloud.

In addition to transaction frameworks that are built as part of a storage system, there has been additional work on frameworks that use various strategies such as restricting transactions to a certain partition, conducting static analysis or scheduling beforehand, or offering an alternate consistency model for transactions. G-Store [11] migrates all keys to a single server before executing the transaction, which requires a transaction to have a priori knowledge of key accesses. Transaction processing systems, such as Lynx [21] and Calvin [20], use static analysis or scheduling to prevent transaction conflicts. In Lynx, transactions are broken down into a series of hops and static analysis is performed in order to determine if the transaction can successfully complete. This enables Lynx to return control to the transaction after the first hop. However, Lynx does reduce transaction completion or preserve transaction ordering. It also only allows application-initiated aborts at the first hop. Furthermore, since chains are static, all participating servers must be known at the start of the transaction. Calvin [20] intercepts transactions in order to perform transaction scheduling. It determines the sequence of transaction execution and applies the same sequence to all of the replicas. Similar to Lynx, Calvin requires a transaction to declare its read and write operations beforehand so that its scheduler can acquire the appropriate locks for the transaction.

Rococo [18] structures a transaction into a collection of atomic pieces, uses a centralized coordinator to aggregate dependency information, and distributes pieces to servers in order to establish an ordering of the operations. By requiring a centralized coordinator, Rococo may not be suitable for geo-distributed transactions with strict latency requirements. Rococo also does not allow user-initiated aborts. Lastly, Sinfonia [7] introduces mini-transactions where operations can be executed within the commit protocol. However, all items that are accessed by a mini-transaction must be known at transaction creation time.

2.3 RPC Chains

CrossStitch's communication pattern is similar to the work found in RPC Chains [19], which chains together multiple RPC invocations to enable computation to flow from server to server. Unlike RPC Chains, CrossStitch provides ACID properties for transactions and does not allow subchains or subtransactions. CrossStitch builds on the approach presented in RPC Chains by organizing transaction servers into a chain while providing transactional guarantees.

3. Architecture

In this section, we detail the construction of a CrossStitch transaction and describe the CrossStitch transaction processing framework and architecture. Furthermore, we present CrossStitch's pipelined atomic commit protocol and demonstrate how CrossStitch overlaps it with transaction execution by interleaving their respective messages. By doing so, CrossStitch is able to reduce the number of cross-datacenter roundtrips and transaction latency.

3.1 A CrossStitch Transaction

CrossStitch is designed to provide support for general transactions. These transactions may have arbitrary dependencies among key operations and may not have a priori knowledge of all key accesses. Unlike a traditional transaction, which consists of operations between `begin` and `commit` statements, CrossStitch transactions are structured as a series of states. Each CrossStitch state is characterized by a key operation and some computation and is executed on a single server.

The states of a CrossStitch transaction form a single chain, where a state in the chain may have a dependency

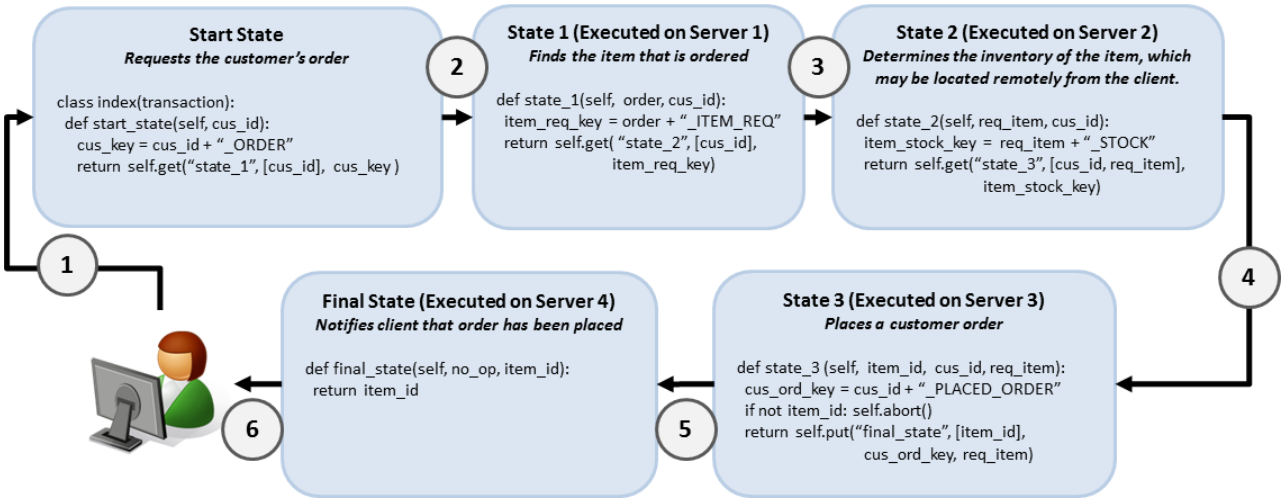

Figure 1: An Example Transaction's Implementation. The transaction is partitioned into multiple states, where each state contains a key access and some computation. We depict the code that is run on each server

on the result of a previous state. CrossStitch state transitions occur when a key operation is encountered, thereby moving the transaction's execution to the server that hosts the requested key. Consequently, a state's execution terminates upon a key operation, and the transaction terminates when a state aborts or encounters a return statement, which contains the result of the transaction, to the client.

To enable computation of CrossStitch states across different servers, CrossStitch servers marshal and send the transaction's implementation, which includes the implementation of all CrossStitch states, and any intermediate data that is required, the server that hosts the transaction's current key operation. Transaction designers may re-order states such that keys, which are co-located at the same geographical location, are adjacent to each other in the CrossStitch transaction chain. This reorganization enables the reduction of cross-datacenter round trips and latency.

3.1.1 An Example Transaction

We demonstrate how CrossStitch executes a transaction using an example that places an order for a customer as illustrated in Figure 1. In this example, the underlying datastore maintains customer information such as the orders that are placed and the items that the customer wishes to purchase. Seller information, such as the inventory of an item, may be located remotely from the customer as the distribution center may be in a different geographical region from the client.

The transaction takes in a customer identification number as a parameter, and begins execution when the client executes the *start_state*, which makes a request for the customer's order information located on $Server_1$. The order information is used in *state_1* to determine the item that the customer is planning to purchase, which requires retrieving the item information from $Server_2$. Before the purchase can be made, *state_2* determines the inventory location and forwards the transaction to *state_3* on $Server_3$ to determine if there is inventory available. Finally, the customer's order is placed on $Server_4$, and the item identifier is returned to the client by *final_state*. In this transaction, the customer information may be in one geographical location, while the inventory and order information may be in different geographical location. For the purpose of illustration, this transaction has been shortened to not include operations for updating the distributor's inventory, and for not returning an order identifier. Adding this to the example requires two additional states.

3.2 Concurrency Control

CrossStitch employs multi-version timestamp ordering for optimistic concurrency control. Every version of an item in CrossStitch's datastore maintains a read timestamp, which is the most recent time object was accessed, and a write timestamp, which is the time when an object was last updated. Both timestamps are used to determine if a transaction needs to abort. Timestamps are specified by the client instead of a time server.

Since timestamps are used to determine the transaction ordering, clocks on CrossStitch clients and servers must be loosely synchronized. Otherwise, large clock skews can lead to a higher abort rate. To mitigate the problem of differing system times, CrossStitch clients and servers can synchronize themselves using NTP [5] or utilize atomic clocks if they are available to reduce spurious aborts due to clock skew.

3.3 Pipelined Commit Protocol

CrossStitch introduces a pipelined commit protocol (PLC) to ensure transactional atomicity. Unlike transaction processing systems that use two-phase commit (2PC) or linear 2PC, PLC does not have a centralized coordinator. In 2PC, the

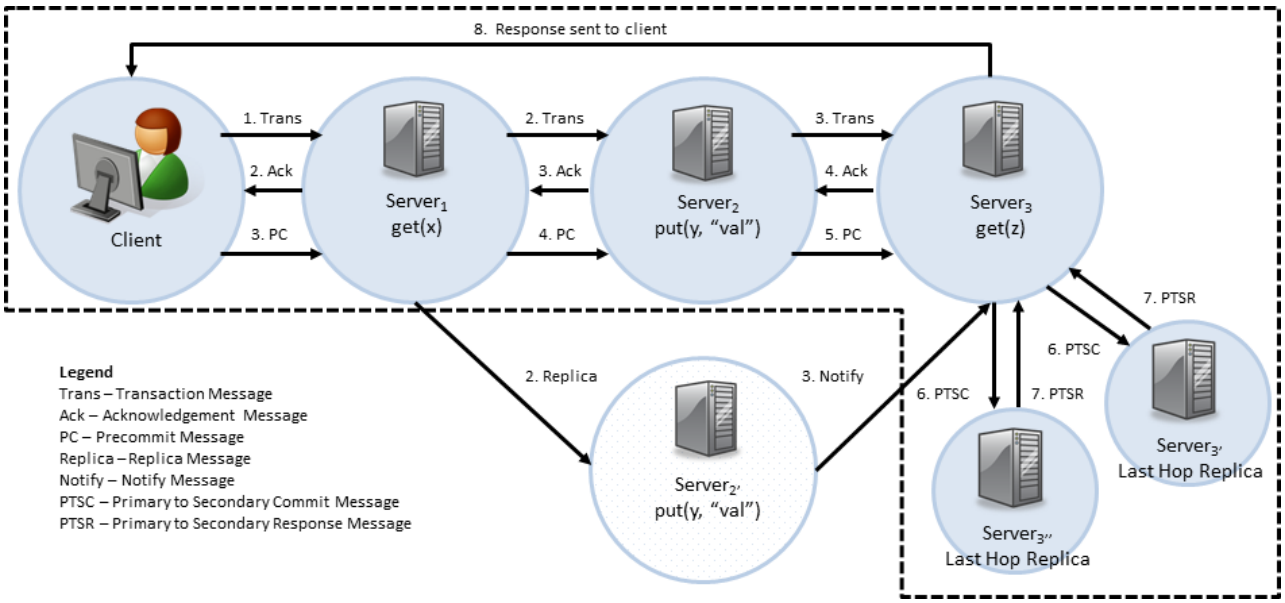

Figure 2: CrossStitch's messaging chain. The figure depicts the different types of messages that are sent between participating servers of the transaction. We describe CrossStitch's messaging pattern and atomic commit protocol in Section 3.3

centralized coordinator sends a vote request message to all participating servers and commits the transaction if and only if all participants decide to commit. Once a server votes to commit a transaction, it cannot later unilaterally abort. Instead of using a centralized coordinator, PLC distributes the role of the coordinator across the participating servers in a transaction. It also forwards the vote commit message sequentially to the participating servers in a manner similar to linear 2PC.

In PLC, a server ($Server_i$) is ready to commit a transaction when it forwards the transaction to the next server ($Server_{i+1}$) as part of a state transition. Upon receiving and processing a transaction message, $Server_{i+1}$ assumes the role of the coordinator and returns an acknowledgement message to the previous server. The acknowledgement message includes the location of $Server_{i+2}$, which allows the transaction to complete in the event that $Server_{i+1}$ fails. A server sends a precommit message, which indicates that it is voting to commit the transaction, when it receives a precommit message from the previous server and an acknowledgement message from the next server. By sending a precommit message, the server indicates that it is delegating the commit decision to the next server. The last server in the chain becomes the final coordinator and performs a variant of Paxos commit [14] with two transaction replicas. Once the Paxos commit completes, it sends a commit or abort message to the client and the participating servers.

The boxed area in Figure 2 illustrates PLC with a CrossStitch transaction with three key operations. It shows the transaction, acknowledgement, and precommit messages between adjacent states. The example also shows the Paxos commit messages between the final server and the transaction replicas.

PLC's design provides transactional atomicity by ensuring that a server cannot unilaterally abort once it has delegated the commit decision to the next server in the chain. Prior to sending the precommit message, a server can abort at any time and the following servers in the transaction chain will eventually timeout waiting on a precommit message and abort the transaction. As an optimization, the server initiating the abort will send *forward abort* messages to subsequent servers in the chain to avoid long timeouts. In the event that $Server_{i+1}$ fails before sending the transaction to $Server_{i+2}$, $Server_i$ will eventually timeout and query $Server_{i+2}$ on the status of the transaction. $Server_{i+2}$ can then abort the transaction if it has not received a precommit message from $Server_{i+1}$. We address the problem of a single server failure causing multiple adjacent states to fail in Section 3.5.1. Additionally, we provide a detailed argument of CrossStitch's liveness and safety in Section 4.

By removing the need for a centralized coordinator and interleaving PLC messages with transaction messages, CrossStitch can provide a chained communication pattern. This can reduce the number of cross-datacenter roundtrips if the chain is partitioned such that the majority of adjacent states are executed in the same datacenter.

3.4 Reducing Cross-Datacenter Messages

CrossStitch's primary objective is to reduce the number of round-trips between geographically distant datacenters. Typically, a transaction processing system with a centralized coordinator requires a significant number of cross-datacenter

round trips since requests for remote data are made through the coordinator. Although these systems have a straightforward transaction implementation, CrossStitch trades off a bit of complexity in transaction implementation in favour of a chained communication pattern, which can reduce the number of cross-datacenter round trips.

In order to reduce the number of cross-datacenter roundtrips using a chained communication pattern, the transaction needs to minimize the number of adjacent states in different geographical locations. This can be done by rearranging states for transactions which have some flexibility in their operation ordering. Many transactions support such rearranging, including the TPC-C-like transactions we use in Section 5 to evaluate the performance of CrossStitch.

A secondary benefit of a chained communication pattern is that, by reducing the number of cross-datacenter operations, sequential operations in a transaction are performed in quick succession, which reduces the opportunity for overlapping transactions to cause dirty reads. As a result, CrossStitch can provide a lower abort rate than traditional transaction processing systems even when the CrossStitch transaction consists of additional operations and the transaction completion time of the systems are equal.

3.5 Adding Replication

In order to provide high availability, we can introduce additional messages to CrossStitch that are used to support replication. Upon encountering a put request at the end of a transaction state, a server will forward the transaction to the server responsible for the put request's key and to the key's replica. The replica will perform the put operation and the computation associated with the next state in order to replicate the put request and determine the next key access in the transaction chain so that a notify message can be sent to the next server. The next server waits until it receives a notification message from the previous server's replica before it can send a precommit message. Figure 2 illustrates both the transaction and replication messages.

This replication scheme, together with PLC, can provide availability with up to one server process failure. We can extend both protocols to support up to k failures by increasing the number of replicas per key to k and the number of transaction replicas to $2k$, and by requiring that a server has received k notification messages if the previous state requested a put operation and k acknowledgement messages from the next k servers in the chain before sending a precommit message.

3.5.1 Adjacent States

In some transactions, adjacent states may be executed on the same server. Consequently, a server failure can cause multiple server processes to fail, which in turn will result in simultaneous failure of adjacent states. In this event, there are cases where PLC would not be able to determine whether or not it can safely commit or abort a transaction. To address this issue, we can group together adjacent states from the same server as a single logical state. The acknowledgement message from a logical state must include the server for the state succeeding the logical state. This will ensure that a single server failure will not result in the simultaneous failure of adjacent states.

Grouping adjacent states on the same server is only sufficient when CrossStitch is configured to only support one server process failure. In order to support k failures, CrossStitch must ensure that, for any sequence of k states, each state is on a different server. To accomplish this, a CrossStitch transaction must introduce additional no-op states on different servers within the same datacenter that are only used to satisfy the aforementioned restriction and provide the required level of availability.

3.6 Design Details

A CrossStitch server must keep track of the messages that it has received and the operations that it has performed for each transaction state in order to correctly handle exceptional cases. These cases include timing out while waiting for a message, receiving an abort message after sending a precommit message, and performing a cascading abort due to a transaction reading uncommitted writes where the uncommitted transaction eventually aborts. Internally, a CrossStitch server is implemented as a state machine where state transitions occur from receiving messages or from timeout events. For the case of handling timeouts, a CrossStitch server will know which servers to contact in order to determine whether it should continue to wait or request a subsequent server in the transaction chain to abort the transaction.

4. Liveness and Safety

In this section, we outline CrossStitch's liveness and safety properties. We demonstrate that CrossStitch transactions either commit or abort (liveness) in the presence of a single server failure and that CrossStitch provides transactional isolation and atomicity (safety). We assume in our liveness and safety argument that servers fail silently. Further details on CrossStitch's liveness and safety properties are found in [13].

4.1 Liveness

We present CrossStitch's liveness properties by showing that all CrossStitch transactions complete, despite the presence of a single, silent failure. We define that a transaction completes when a final result is determined and the transaction commits or aborts. In order to ensure that the appropriate messages have been received by a server for it to delegate its commit decision to the subsequent server in the transaction chain, all CrossStitch servers maintain *internal state*, which we depict in Figure 3. The internal state of a server refers to a server's status regarding the PLC messages that it has received, as opposed to a CrossStitch state, which consists of a

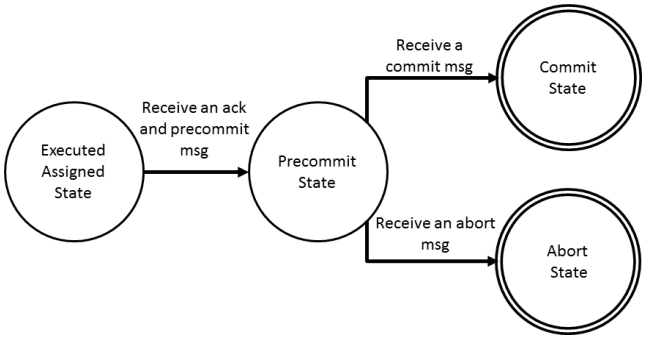

Figure 3: CrossStitch's internal states. Upon receiving and executing a transaction, a server has executed its assigned state. A server delegates the commit decision and enters the precommit state when receives an acknowledgement and a precommit message. A server can no longer unilaterally abort once it is in precommit state.

key operation and computation. CrossStitch utilizes message metadata, such as knowledge of other servers in the transaction chain, and timeout mechanisms to determine server failures and to also determine if a transaction commits or aborts. As CrossStitch servers maintain the internal state of each executed transaction state, they can determine which other servers to query in the event of a failure.

Once a server receives and executes a transaction, it has executed its assigned transaction state and waits for the acknowledgement and precommit messages before it passes the role of coordinator to the next server in the transaction chain. We say that a server is in *precommit state* if it has executed its assigned transaction state, received an acknowledgement message and a precommit message, and delegated the commit decision to the subsequent server in the transaction chain. A server enters the *commit state* when it has received a commit message and the *abort state* when it has received an abort message.

We now demonstrate CrossStitch's liveness property for a single, silent server failure. We begin by considering that $Server_i$ is in *precommit state*. As mentioned in Section 3.6, once a server is in *precommit state*, it cannot abort the transaction. In order for $Server_i$ to be in *precommit state*, it must have received a precommit message from $Server_{i-1}$ (hence $Server_{i-1}$ must also be in precommit state) and an acknowledgement message from $Server_{i+1}$. For $Server_{i+1}$ to have sent an acknowledgement message, it must have executed its assigned transaction state; thus, $Server_{i+1}$ has knowledge of $Server_{i+2}$. $Server_i$ receives metadata that indicates that $Server_{i+2}$ is two servers forward in the transaction chain; thus, $Server_i$ can now query $Server_{i+2}$ in the event that the transaction times out.

We now suppose that $Server_{i+1}$ fails silently. It is uncertain if $Server_{i+1}$ completed executing its corresponding transaction state and/or has sent its acknowledgement message. If $Server_i$ does not receive an acknowledgement message after a predetermined timeout, $Server_i$ can safely abort the transaction since it is not in *precommit state*. However, if $Server_i$ is in *precommit state*, it can query $Server_{i+2}$ to determine the state of the transaction. If $Server_{i+2}$ has not received the transaction or is not in *precommit state*, $Server_{i+2}$ can abort the transaction on $Server_i$'s behalf as $Server_{i+2}$ has not entered *precommit state*.

However, if $Server_{i+2}$ is in the *precommit state*, then this indicates that the transaction has progressed, and $Server_i$ should now wait for the commit message from the final server in the transaction chain. Before committing the transaction, the final server in the transaction chain, which is acting as the final coordinator, will attempt to notify $Server_{i+1}$ that the transaction has committed. If $Server_{i+1}$ is unavailable, the keys that $Server_{i+1}$ is hosting will failover to $Server_{i+1}$'s replica. Upon recovering from its failure, $Server_{i+1}$ will check with other servers in the transaction chain regarding the status of the transaction and will also synchronize itself with its replica before resuming its original role. Therefore, all transactions either commit or abort if an intermediate server fails.

In the event that the end server ($Server_n$) fails, the preceding server ($Server_{n-1}$) may query the transaction replicas to determine if the transaction should commit or abort. The transaction replicas will follow the Paxos commit protocol to determine if the transaction commits or aborts.

4.2 Safety

We now show that CrossStitch provides transactional isolation and atomicity. As mentioned in Section 3.2, CrossStitch uses multiversion timestamp ordering, which ensures that CrossStitch transactions provide serializable isolation. CrossStitch provides atomicity since CrossStitch servers in a transaction chain do not have differing final internal states. Suppose two servers in the transaction chain are in different final internal states. For a server to be in *commit state*, the end server ($Server_n$) must have successfully completed the transaction. By CrossStitch's specification, all previous servers in the transaction chain must have been in the *precommit state*. Therefore, no server in the transaction chain could have aborted; otherwise, at least one server in the transaction chain would not have been in the *precommit state*. For a server to be in the *abort state*, it must have either received an abort message or have aborted the transaction. If a server receives an abort message, then at least one other server in the transaction chain must have aborted the transaction. Consequently, the server that initiates the abort cannot be in *precommit state* as it did not successfully execute its transaction state or performed a user abort. Therefore, CrossStitch is safe as no two servers in a transaction chain can be in differing final internal states.

5. Evaluation

We evaluate CrossStitch against a traditional transaction processing system by comparing transaction completion time, throughput, and abort rates of synthetic transactions which consist of a mixture of sequential read and write operations, and TPC-C-like transactions.

5.1 Experimental Setup

Our experimental setup consists of four machines when simulating a single or two datacenter environment and six machines when simulating a three datacenter environment. Each machine has two 2.10GHz Intel(R) Xeon(R) CPU E5-2620 v2 CPUs and 64GB of RAM. For our experiments that simulate geographically distant datacenters, we partition our machines into groups of two, where each group represents a single datacenter. Multiple server instances are executed on one machine, and clients are executed on the other. We use *tc* [6], a network traffic control and shaping application for Linux, to add latency between machines to simulate geographically-distant datacenters.

In addition to CrossStitch, we implemented a traditional transaction processing framework based on a flat transaction processing system model that uses 2PC to ensure transactional atomicity; we refer to this framework as the 2PC system. When executing a transaction in the 2PC system, a client first selects a coordinator and then sends all read and write requests to the coordinator. The coordinator relays the read and write requests to the appropriate servers and keeps track of the transaction participants. To allow for greater concurrency, read operations immediately return future objects; thus, a read operation does not block until the result of the read is needed. Write operations are asynchronous and require the client to store a local copy of updated values that have not been committed. Similar to CrossStitch, our 2PC system replicates write operations to ensure that each value is stored on two servers.

To allow for a fair evaluation, our 2PC system and CrossStitch share much of the same code base, including their concurrency control mechanism and their key-value datastore. Both systems use multi-version timestamp ordering; therefore, the difference in abort rate between these systems is not the result of the implementation or the type of concurrency control.

5.2 Synthetic Workload

We first use a synthetic workload to determine the performance characteristics of CrossStitch and our implemented 2PC system. We compare their performance in a single datacenter deployment to quantify the overhead of forwarding a transaction from server to server, and also in a multi-datacenter deployment to determine the performance impact of wide-area network latency on the two systems.

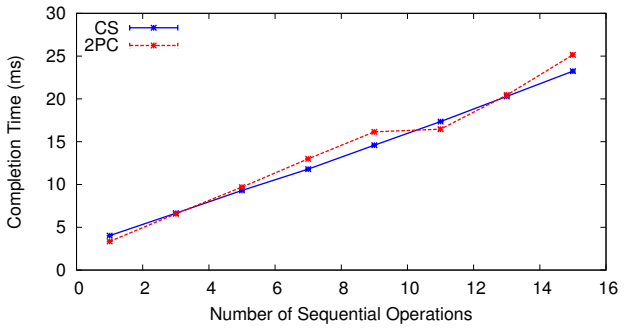

Figure 4: Completion Time vs. Chain Length. Each transaction consists of 80% read operations

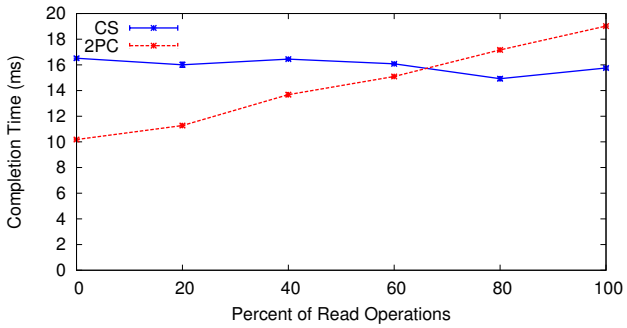

Figure 5: Completion Time vs. Percentage of Read Operations

5.2.1 Single Datacenter Evaluation

To quantify the overhead of forwarding a transaction across servers in a chain communication pattern, we first evaluate CrossStitch and our 2PC system in a single datacenter environment. In order to evaluate general transactions with dependencies between key operations, our synthetic transactions are characterized as a sequence of dependent read or write operations. In this workload, keys are four byte strings, values are 1 KB strings, and key popularity follows a uniform random distribution. Unless otherwise stated, each transaction consists of 80% read and 20% write operations and contains a total of ten operations.

In Figure 4, we vary the number of operations per transaction. As expected, as the number of operations increase, the completion time of CrossStitch and the tradition system increase linearly. CrossStitch introduces a small increase in completion time per operation due to the overhead of marshalling and unmarshalling the transaction's data and implementation.

In the next experiment, we evaluate CrossStitch's performance with different percentages of read operations in a ten operation transaction. Figure 5 shows that CrossStitch's performance is largely independent of the read percentage. This is because CrossStitch's replication messages are sent in par-

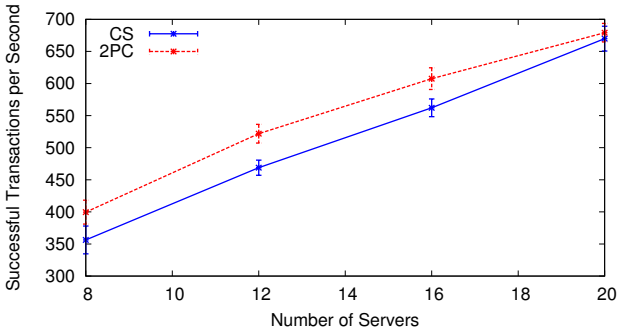

Figure 6: Completed transaction throughput of CrossStitch and 2PC in a single datacenter environment

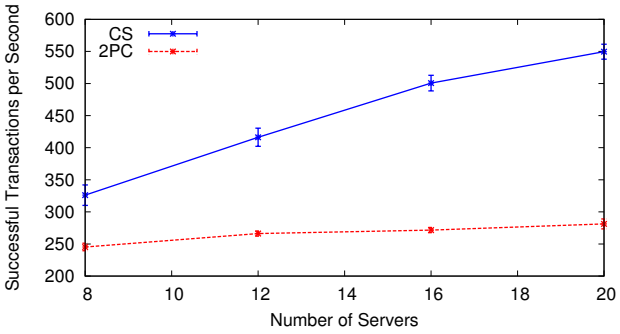

Figure 7: Completed transaction throughput of CrossStitch and 2PC in a two-datacenter environment with 50 ms latency

allel with the transaction messages, which makes writes only marginally slower than reads since the next server must still wait until it receives a message from each replica. In contrast, reads are significantly faster than writes for the 2PC system because writes are asynchronous due to client-side buffering where reads are synchronous because of order dependencies. For transactions that consist of mostly read operations, CrossStitch has similar performance to our 2PC system even in a single datacenter environment.

5.2.2 Throughput and Scalability

To compare the throughput and scalability of CrossStitch and our 2PC system, we determine the maximum number of successful transactions per second that each system can support for 8 to 20 servers. To do so, we increase the number concurrent clients until we saturate the system.

Figure 6 shows that the completed transaction throughput of CrossStitch shows similar scaling properties to our 2PC system when the servers are all in the same datacenter. The results show that CrossStitch has a completed transaction throughput that is slightly less than our 2PC system. This is likely due to the additional overhead of marshalling/unmarshalling the data and transaction implementation. The overhead is more pronounced in this experiment than in the previous single client experiment because overhead from simultaneous transactions can compound.

We also evaluate CrossStitch's scalability for geo-distributed transactions. For this experiment, four machines are partitioned into groups of two and *tc* is used to add 50 ms of latency between each group of machines. One machine in a group is responsible for running the server processes while the other executes the clients. Transactions are structured such that the first set of operations are executed at the same datacenter as the client, and the remaining operations are executed at a remote datacenter. CrossStitch will therefore require two cross-datacenter roundtrips to process each transaction. In contrast, the traditional transaction system requires five cross-datacenter roundtrips on average. Four round trips are attributed to remote read operations, and one round trip is attributed to the commit operation. As

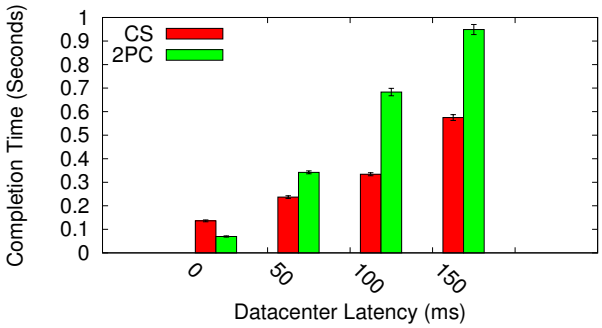

Figure 8: Completion Time vs. Datacenter Latency

a result, when processing a transaction, the traditional system incurs more latency. Figure 7 shows that CrossStitch has a higher throughput than our 2PC system. More importantly, CrossStitch's throughput scaling is largely unaffected by the additional 50 ms latency. In contrast, our 2PC system's throughput does not scale with more servers with the additional 50 ms latency. We analyze the reason for this behaviour in the next section.

5.2.3 Increasing Latency Between Datacenters

To evaluate the performance of CrossStitch in a geo-distributed environment with different latencies between datacenters, we repeat the same experiments in Section 5.2.2 but with latencies from 50 to 150 ms and fix the number of servers to eight. We again assume that data accesses in a transaction are organized so that key operations on the same datacenter are grouped together, and that the client performs local key operations before performing remote operations.

Figure 8 shows the effect of cross-datacenter latency on transaction completion time. The latency indicated on the x-axis is the round-trip between the two simulated datacenters. Since CrossStitch transactions require two cross-datacenter round-trips, the latency of a CrossStitch transaction is always twice of the cross-datacenter roundtrip latency. For our 2PC system, since transactions are 80% read operations and

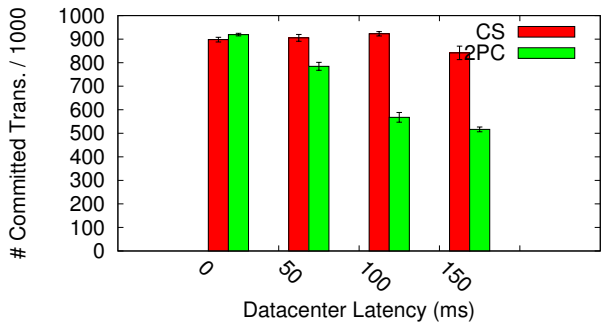

Figure 9: Number of Successful Transactions vs. Datacenter Latency

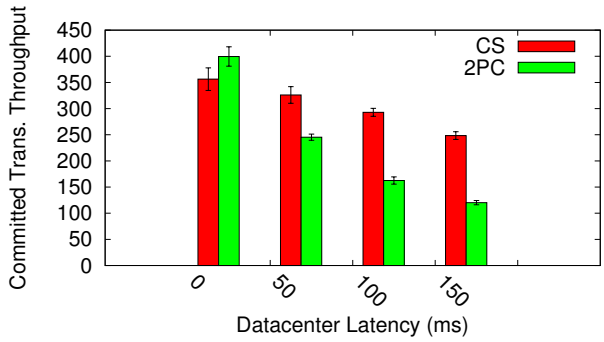

Figure 10: Completed Transaction Throughput vs Datacenter Latency

there is an equal number of servers at each datacenter, it must perform, on average, four remote read requests. Since read operations are not asynchronous, each read will require one round trip. As a result, each two-phase commit transaction requires five round trips (four for read operations and one for the commit protocol). Therefore, as datacenter latency increases, the completion time increase at a much higher rate than CrossStitch. Note that in this experiment, unlike in previous latency experiments, we use enough clients to saturate the system instead of just a single client.

Since CrossStitch transactions have a lower completion time than 2PC transactions in a geo-distributed environment, there is a smaller window for overlapping operations from different transactions to occur. We find that a large percentage of the aborts are a result of write operations causing pending read operations to be invalid. Many aborts are also due to cascading aborts. A cascading abort occurs when an uncommitted transaction's read operation reads another uncommitted transaction's write operation; however, the transaction of the pending write operation aborts, causing the dependent read operation to be invalid. Figure 9 shows the number of successful transactions (out of 1000) compared to the latency between datacenters. Because CrossStitch transactions complete more quickly and have less operations that overlap between transactions than our 2PC system, it has a

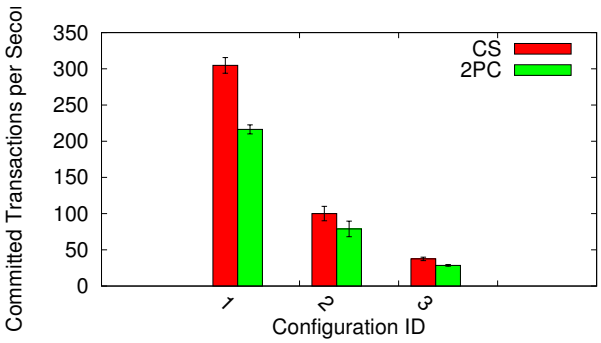

Figure 11: Successful Transaction Throughput vs. Datacenter Configuration. Datacenter latencies used are 1:(70ms, 40ms, 50ms), 2:(140ms, 164ms,308ms) and 3:(463ms, 546ms, 318ms)

significantly higher completion rate than our 2PC system. The transaction completion rate for CrossStitch remains constant until the cross-datacenter latency is 150 ms. As a result of the higher transaction completion rate, the throughput of completed transactions for CrossStitch is also greater than the 2PC system as shown in Figure 10. The drop in throughput for CrossStitch is due to the increase in the number of concurrent transactions due to the higher cross-datacenter latency, which results in greater overhead.

5.2.4 Three Datacenters

We also evaluate CrossStitch and the traditional transaction processing system across three datacenters. For this experiment, we use a similar setup to that found in Section 5.2.3; however, we use additional machines for servers and clients.

Our evaluation includes three experimental configurations that represent geographically distributed locations. Our first configuration simulates datacenters located in New York (USA), California (USA), and Texas (USA). The second configuration uses London (UK), California (USA), and Hong Kong. Lastly, the third configuration simulates datacenters that are located in Brisbane (Australia), Shanghai (China), and Paris (France). The latency between these locations were determined from measurements taken on December 25, 2014 in [3]. For each of the configurations, the latencies that are used are (70ms, 40ms, 50ms), (140ms, 164ms, 308ms) and (463ms, 546ms, 318ms) respectively.

Again, we sort our operations such that local operations are performed before remote operations. As shown in Figure 11, CrossStitch's improvement over our 2PC system is relative to the latency between the datacenters. In all cases, CrossStitch has higher throughput than our 2PC system.

5.3 Evaluation Using TPC-C-like Transactions

In addition to the results presented in Section 5.2, we compare CrossStitch with our implemented 2PC system using more realistic transactions that involve sequential and dependent key operations. Specifically, we evaluate CrossStitch

and our 2PC system using two TPC-C-like transactions. These transactions are based on the order status and stock level transactions. The order status transaction reads all order lines for a customer's order information, and the stock level transaction counts all stock in a district that is below a given threshold. We first perform a worst case theoretical analysis for both CrossStitch and our 2PC system.

5.3.1 Theoretical Analysis

In our evaluation, the keys and values used in our transactions are specified in the TPC-C-guidelines. Keys can consist of multiple fields, and we partition our data across datacenters based on the these fields. Warehouse keys are partitioned according to the warehouse identifier. Keys that pertain to history, district, customers, orders, new orders and orderlines are partitioned according to their district identifier. Lastly, items and stock are partitioned according to their item identifier. Although stock information may be partitioned on the warehouse identifier, our evaluation uses a single warehouse. Therefore, we partition the stocks on item identifiers to achieve better load balancing across servers. In our experiments, we partition our keys in across two datacenters.

To demonstrate that CrossStitch has the potential to significantly improve upon a traditional 2PC system, we perform a worst-case analysis on the roundtrip latencies that a transaction incurs. In our analysis, we assume that the client is located at a different datacenter from the data. Furthermore, for the 2PC system that utilizes two-phase commit, the client and coordinator are co-located on the same datacenter.

We begin by considering the order status transaction which retrieves customer information, followed by the customer's order, and finally all order lines. The operations in the order status transaction are dependent on one another. Consequently, the 2PC system would require at least three round trips to obtain the data plus an additional round trip for the commit protocol. On the other hand, the CrossStitch client sends the transaction to the remote datacenter and the servers of the remote datacenter perform the transaction and send the result to the client. As there is a round of acknowledgement and precommit messages sent between the client and the first server in the transaction chain, we include the expected round trip time that includes the commit protocol CrossStitch incurs two round trips.

Similarly, the stock level transaction retrieves a district, orders that pertain to the retrieved district, order lines of orders that are read, and the stock associated with the order lines. In this transaction, there are four read operations that are dependent on each other. Therefore, if the client was located on a remote datacenter, the 2PC system would require five round trips whereas CrossStitch will require two round trips. The results of this analysis are summarized in Figure 12.

Transaction	Traditional	CrossStitch
Order Status	3+1	2
Stock Level	4+1	2

Figure 12: Worst case round trip comparison for a traditional system vs. CrossStitch

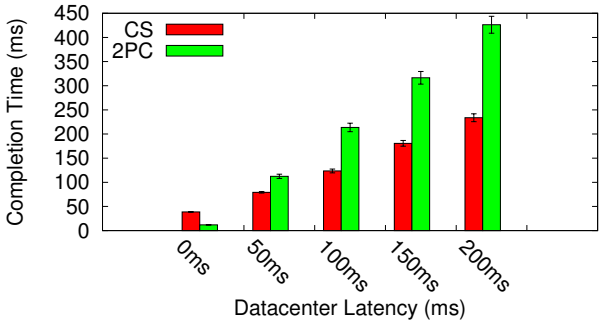

Figure 13: The average completion time of the ORDER STATUS transaction as datacenter latency varies

5.3.2 Results of TPC-C-like Transactions

In this experiment, we compare the transaction completion time for the order status and stock level transaction using CrossStitch and the 2PC system. Figure 13 shows the transaction completion time of the order status transaction for CrossStitch and the 2PC system where we vary the datacenter latency. The results demonstrate that CrossStitch requires two fewer round trips than the traditional system when completing the order status transaction. As datacenter latency increases, the improvement that CrossStitch provides is more evident. We also consider the case where there is no latency between the two datacenters. We see that CrossStitch has a higher average transaction completion time when there is no datacenter latency, which is attributed the CrossStitch's overhead of marshalling, sending, and unmarshalling a transaction. As datacenter latency increases, the 2PC system incurs more latency than CrossStitch, demonstrating that incurring CrossStitch's overhead latency is worthwhile is such situations.

In the case of the stock level transaction, we reduce the number of inventory items that are retrieved in order to not overwhelm the servers. We found that a single transaction sends upwards of a couple hundred of key requests, many of which are concurrent. Given that our server deployment for both CrossStitch and the traditional system consists of eight servers, these servers become inundated with requests, thereby causing the transaction completion time for our 2PC system to be significantly higher than expected. As a result, instead of retrieving the entire stock for the last 20 orders in a district, we retrieve the stock of a single item from a single order in a district and a single order line of an order. By doing so, our servers are not overrun with key opera-

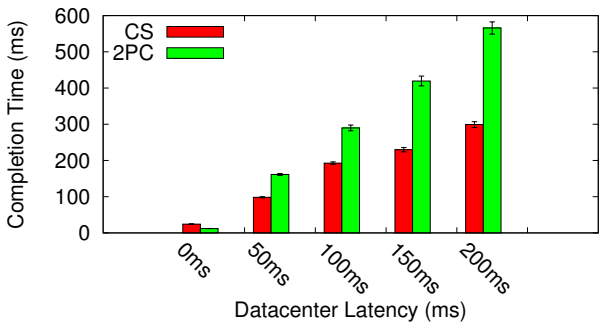

Figure 14: The average completion time of the STOCK LEVEL transaction as datacenter latency varies

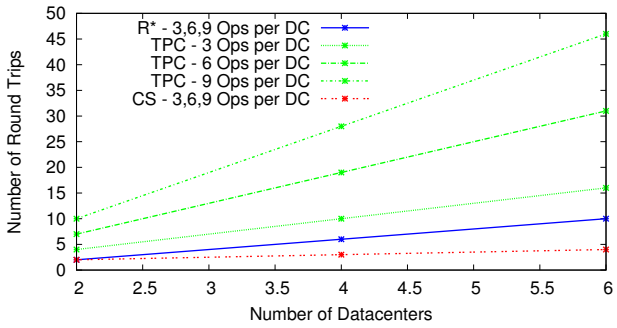

Figure 15: The number of round trips incurred as the number of datacenters travelled increases

tions, and the transactions in the traditional system complete in our expected time. We also change the stock level transaction for CrossStitch in this manner to provide a fair comparison between the two systems. As shown in Figure 14, CrossStitch provides a significant improvement over the 2PC system as was expected by our analysis in Figure 12. We note that a stock level transaction might not require any cross-datacenter trips, resulting in a lower completion time than our worst-case analysis. Nonetheless, Figure 14 demonstrates that CrossStitch provides lower latency than our 2PC system.

5.4 Analytical Comparisons

In our evaluation, we implemented a flat transaction processing system that uses a centralized coordinator to send key requests and perform the commit protocol. Improvements upon our implemented flat transaction processing system includes the system presented in the R* System [17], where the commit protocol is executed in a tree-like fashion. In this system, the root node of the commit tree acts as the coordinator, leaf nodes are participants, and the remaining nodes serve as both coordinator and participants. The vote of a node depends on the vote of its children and its part of the transaction.

An improvement upon the flat transaction processing system is to use code shipping, where code and key requests are delivered to the remote datacenter, with the R* commit protocol. Such a transaction processing system would require only one round-trip for a group of remote key accesses, as opposed to one round-trip for every remote key access.

As an R*-like system with code-shipping was not available to us, we analytically determine the expected number of round trips that are required to complete a transaction. In our analysis, we assume that local key accesses and computation are small compared to cross-datacenter round-trip time; thus, we only consider cross-datacenter round-trip time in our analysis. We analyze synthetic transactions that consist of multiple, sequential key accesses. We compare the round trips incurred by our implemented, flat transaction processing system, a system that uses code shipping and the R* commit protocol, and CrossStitch.

All operations in our analyzed transactions are sequential and occur one after another. In Figure 15, we depict the relationship between the number of datacenters that the transaction contacts and the number of round trips that transaction execution will incur. The various lines indicate the number of operations that occur per datacenter. In the case of CrossStitch and code-shipping with the R* commit protocol, as operations are done locally, cross-datacenter latency is incurred per group of operations as opposed to each time a key is accessed. However, as the number of key accesses on remote datacenters increases, the flat transaction processing system incurs significantly more round trips.

As CrossStitch performs the pipelined commit protocol during transaction execution, the latency that is incurred by CrossStitch as a result of cross-datacenter round trips is less than a transaction processing system that uses code-shipping and a R* commit protocol. The difference in latency between CrossStitch and a system that uses code-shipping and the R* commit protocol becomes greater as the number of remotely accessed datacenters increase. As shown in Figure 15, if a transaction accesses data on six remote datacenters, CrossStitch incurs four round trips while the transaction processing system that uses code shipping and the R* commit protocol incurs ten round trips.

6. Conclusion

In this paper, we have presented CrossStitch, an efficient transaction processing framework for geo-distributed datastores. We described the CrossStitch transaction structure and architecture, and we presented CrossStitch's pipelined atomic commit protocol that executes in parallel with the transaction, thereby masking commit latency with execution latency and preserving a chained communication pattern across servers. We also provided detailed liveness and safety arguments for CrossStitch. Our performance evaluation showed that CrossStitch performs significantly better a traditional transaction processing system using 2PC.

References

[1] Big data: Are you ready for blast-off? http://www.bbc.com/news/business-26383058.

[2] Distributed recovery. http://research.microsoft.com/en-us/people/philbe/chapter7.pdf.

[3] Dot-com monitor - network latency. https://www.dotcom-tools.com/internet-backbone-latency.aspx.

[4] Lightweight transactions in cassandra 2.0. http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0.

[5] Ntp: The network time protocol. http://www.ntp.org/.

[6] Traffic control howto. http://tldp.org/HOWTO/Traffic-Control-HOWTO/.

[7] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[8] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research*, pages 223–234, 2011.

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

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[11] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the First ACM Symposium on Cloud computing*, 2010.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[13] A. for submission. Anonymized for submission. Master's thesis, Anonymized for submission, 2014.

[14] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[15] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[16] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[18] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the eleventh USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[19] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. Rpc chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the Sixth USENIX Symposium on Networked Systems Design and Implementation*, pages 277–290, 2009.

[20] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[21] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.