Efficient Transaction Processing for Short-Lived Transactions in the Cloud

by

Sharon Choy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The cloud, in the past few years, has become the preferred platform for hosting web applications. Many of these web applications store their data in a distributed cloud storage system, which greatly simplifies application development and provides increased availability and reliability. However, with increasing user demand for web applications, these cloud storage systems often become the performance bottleneck. To address the cloud's performance demands, many storage system features, such as strong consistency and transactional support, are often omitted in favour of performance. Nonetheless, transactions remain necessary to ensure data integrity and application correctness.

In this thesis, we introduce CrossStitch, which is an efficient transaction processing framework for distributed key-value storage systems. CrossStitch supports general transactions, where transactions include both computation and key accesses. It is specifically optimized for short-lived transactions that are typical of cloud-deployed web applications. In CrossStitch, a transaction is partitioned into a series of components that form a transaction chain. These components are executed and the transaction is propagated along the storage servers instead of being executed on the application server. This chained structure, in which servers only communicate with their immediate neighbours, enables CrossStitch to implement a pipelined version of two-phase commit to ensure transactional atomicity. CrossStitch is able to eliminate a significant amount of setup overhead using this structure by executing the transaction and the atomic commit protocol concurrently. Therefore, CrossStitch provides low latency and efficient transactional support for cloud storage systems. Our evaluation demonstrates that CrossStitch is a scalable and efficient transaction processing framework for web transactions.

Acknowledgements

Firstly, I would like to thank my supervisor Prof. Bernard Wong for his guidance, support, and patience during the course of my Master's degree program and in the preparation of this thesis. I would also like to thank my thesis readers Prof. Ken Salem and Prof. David Taylor. I greatly appreciate their helpful comments and feedback as their insight has allowed me to significantly improve the quality of this thesis. I am thankful for Xiaoyi Liu who helped me with the experimental evaluation of CrossStitch. Furthermore, I am grateful for the friendships that I have formed with all the members of the Shoshin Lab. I am blessed to have such caring and supportive brothers and sisters at Kitchener-Waterloo Chinese Alliance Church. Finally, I am forever thankful and grateful for my family (my mother - Marie Choy, my father - Robin Choy, and my brother - Ehren Choy) as their unconditional love and encouragement have brought me here today.

Dedication

"So whether you eat or drink or whatever you do, do it all for the glory of God."
- 1 Corinthians 10:31 (NIV)

Table of Contents

List of Figures

Chapter 1

Introduction

In recent years, web applications have emerged as a popular alternative to traditional client-side applications. The rise in popularity of web applications is partially due to the pervasiveness and availability of web browsers, which greatly reduce the barrier to deploying web applications by eliminating the need to download a separate application binary. Users are able to access web applications without concerning themselves with security patches or software upgrades. Web applications offer end-users platform independence as they can be deployed on any web browser and also on any computer. Their popularity has resulted in a demand for highly scalable infrastructure that is able to service a large number of concurrent users. Furthermore, these applications must serve client requests quickly as previous studies [11] have demonstrated that even a small increase in access latency can significantly reduce the user traffic volume.

The scalability and performance requirements of web applications have made the cloud the preferred application hosting platform for most developers. Web applications running in the cloud can quickly increase their resource allocation, which can be used to run additional web or application servers, to handle increased client demand. However, these applications may still experience scalability problems when accessing and updating shared data in a cloud storage system. For example, the storage mechanism responsible for hosting a shopping cart application must remain highly available and allow consumers to add and remove items efficiently. As a result, many NoSQL systems, which provide their end users with simple key-value interfaces, have been developed in response to the need for high scalability and performance. These storage systems trade off rich functionality in favour of performance. For example, features such as strong consistency, which ensures that all replicas are up-to-date, are often omitted. This concession is made in order to improve performance for applications that can accept weaker consistency models.

In addition to the omission of a strong consistency model, many cloud storage systems also lack support for general transactions. Distributed transactional support has traditionally been complicated and expensive due to the coordination that is required between different servers. The required coordination results in many message round trips between participating servers. Therefore, providing transactions results in additional system complexity that hinders performance. Nonetheless, transactions remain critical for many web applications because they are needed to ensure the correctness of the application and the integrity of the storage system. Currently, since many commercially available NoSQL systems do not offer support for transactions, most applications requiring transactions implement transactional support at the application layer; thus, the burden of providing transactions rests on the application developer. This method for providing transactions is error prone since the application's implementation may be done poorly, inefficiently, or in an ad-hoc manner. Alternatively, developers may use existing cloud storage systems that provide transactional support; however, these systems have significant limitations. For example, some systems (e.g., [12]) require that all key accesses be within a certain range. Other systems may experience significant latency for certain operations. For example, Megastore [9] has a write latency of 100 ms-400 ms, whereas its read latency is in the tens of milliseconds.

1.1 Towards a Framework for Supporting Short-Lived Transactions in the Cloud

We consider characteristics of typical transactions found in web applications in order to build an efficient transaction processing framework. Moreover, we need to consider the performance requirements for hosting such a framework in the cloud.

1.1.1 Characteristics of Transactions in Web Applications

Most web transactions exhibit the following properties:

- **Web transactions are short-lived:** A typical web transaction must complete quickly (i.e., on the order of milliseconds). These transactions generally access and update only a few items in the datastore. Consider the example of an e-commerce web application where a buyer wishes to purchase an item from the seller. A transaction is necessary for the payment of the items to ensure that a debit can occur (i.e., the

buyer has enough funds to purchase the item) and that the seller's account is credited. Because a web transaction is short-lived, the overhead in setting up the transaction may account for a significant fraction of the transaction's execution time. Therefore, a transaction processing system for web transactions should aim to minimize this overhead.

- **Most web transactions are executed sequentially:** Many key accesses in a web transaction depend on the current state of the application or on computation that depends upon the result of previous key accesses. We consider an application that adds interest to a bank account. The application must first retrieve the account type and country of residence in order to determine the interest rate. Afterwards, the application obtains the current account balance, calculates the new balance, and writes the new value into the datastore. Another example is a data indexing application that uses transactions to ensure data consistency and integrity. In this application, an index is keyed on a secondary data attribute, called the secondary key, and each index entry contains the primary keys of data items that share the same secondary key. Accessing a data item using an index requires an index lookup followed by at least one primary key lookup. These lookups are dependent on each other and must be performed sequentially. Therefore, a transaction processing system for web applications must efficiently support transactions that execute sequentially and determine their key accesses dynamically rather than at the start of the transaction.

- **Web transactions may request data that is distributed across many datacenters:** Client data may be partitioned and distributed across multiple datacenters. Therefore, a transaction processing framework must be able to handle transactions that span more than one datacenter. Additionally, application servers and storage servers may be geographically distant from each other, or they may be in different parts of the same datacenter network; thus, these servers may experience additional latency when communicating with each other. Therefore, a transaction processing framework should also minimize the number of round-trips between distant servers.

1.1.2 Requirements for a Transaction Processing System

In addition to considering the characteristics of web transactions, a transaction processing system must also account for the cloud's performance requirements. Firstly, it must be scalable and handle workloads that are typically found in a web application. These workloads include serving many end-users concurrently. Existing commercial databases (e.g.,

[3, 5, 6]) require a coordinator to setup and finalize a transaction. The use of a transaction coordinator often results in a bottleneck at a single node, which could hinder the scalability of a transactional system. Therefore, an efficient transaction processing system should ideally avoid having a centralized coordinator.

Secondly, a transaction processing framework needs to be highly available, as a service outage can result in a significant loss of revenue for the application developer. Therefore, such a system must be able to handle node failures and its transactions must be able to make forward progress (i.e., determine if the transaction needs to commit or abort) despite the presence of such failures. In traditional database systems, servers detect failures and ensure that a transaction does not make forward progress in the event of a failure. Moreover, data is synchronously written into persistent storage in order to provide durability, which enables data to be retrieved and the failed server to recover. In these systems, data may not be replicated across multiple servers. Although such a system provides durability, it does not provide high availability as a transaction will need to wait for a failed server to recover. Conversely, NoSQL systems use replication as an alternative to synchronous writes, which eliminates the waiting time required for data to be written to persistent storage. Most cloud storage systems provide high availability and reliability through replication and do not provide durability through synchronized writes to persistent storage. Therefore, a transaction processing framework that is hosted in the cloud should provide similar reliability and availability guarantees as cloud storage systems.

Lastly, a transaction processing framework must be able to provide low latency for its end-users since latency significantly affects end-user experience. As stated in [11], a latency that is greater than $100ms$ may cause a customer to seek alternative service providers. Since the transaction processing framework is deployed in the cloud, we may need to consider a scenario where the application server is geographically distant from the datacenter that hosts the storage servers. In traditional transactional systems, there is substantial communication between the application server and storage servers. Therefore, if application servers and storage servers are geographically distant from each other, there may be significant delay in completing the transaction.

1.2 CrossStitch: An Efficient Transaction Processing Framework for Short-lived Transactions

In this thesis, we introduce CrossStitch, which is an efficient transaction processing framework for distributed key-value storage systems that is optimized for web transactions.

CrossStitch supports general transactions for key-value stores, where transactions include computation and key accesses. CrossStitch transactions do not require the transaction writer to know all key accesses prior to the execution of the transaction. Moreover, key requests do not have any restrictions on their physical location or on any other characteristic. For example, CrossStitch key requests are not limited to a single node or a single datacenter, whereas other systems such as G-Store [20] require all keys to reside on a single node.

CrossStitch introduces a novel execution method where a transaction is partitioned into a series of components that form a chain. These components are executed on storage servers instead of application servers. The transaction comprises a series of components that form a chain and is propagated from server to server. Each CrossStitch component is characterized by a key request and application-defined computation. Each component also takes in as a parameter the result of the previous component as an input parameter. For each occurrence of a key request, the execution transitions from one component to another. And during each transition, the key request and the transaction's implementation code is delivered to the key's storage server and executed.

In a traditional transaction processing system, clients (e.g., application servers) are responsible for registering the transaction with a transaction coordinator, for issuing key requests to servers, and for notifying the transactional coordinator once the transaction has completed all of its key requests and is ready to commit. Unlike a traditional transaction processing system, a CrossStitch client is not responsible for requesting every key access from remote servers and performing the computation. Instead, a CrossStitch client sends its transaction to the server that hosts its first key access. Afterwards, each CrossStitch server executes a single component in the transaction chain and forwards the transaction to the next server that is responsible for the key request. The final server sends the result to the client. As a result, CrossStitch eliminates multiple round trips between the client and servers. Therefore, since there may be significant delay between application and storage servers due to geographical distance, CrossStitch is able to reduce the amount of latency that is normally incurred by traditional transaction processing systems.

By having a chained structure for a CrossStitch transaction, we can perform additional optimizations such as pipelining the traditional two-phase commit protocol and eliminating the need for a transaction coordinator. Upon receiving a transaction, each CrossStitch server executes its assigned component and queries the previous server (or the client) in the transaction chain to determine if the previous server can commit. If a CrossStitch server can confirm that the previous server can commit and that the next server has received the transaction, it enters a precommit state, which indicates that it is ready to commit. When the final server completes the transaction, it notifies all participating servers to commit the

transaction's operations. By performing the two-phase commit protocol in line, CrossStitch introduces minimal latency to the transaction's execution.

We show the liveness and safety properties of CrossStitch's pipelined two-phase commit protocol by providing an exhaustive enumeration of execution states. We first consider the case where there are no server failures in a CrossStitch transaction chain. We show that a CrossStitch transaction always progresses to completion when there are no server failures. We further extend this argument to show that a CrossStitch transaction terminates in the presence of a single server failure. This thesis makes three contributions:

- **Low-latency for cross-datacenter transactions:** CrossStitch's execution pattern eliminates the use of a transaction coordinator and thereby eliminates the setup overhead that occurs in a traditional transaction. Moreover, CrossStitch's chained-message structure reduces messaging between application servers and storage servers. By eliminating setup overhead and reducing the number of messages, we can significantly reduce the latency of transactions. These performance improvements are significant in an environment where a transaction spans multiple, geographically-distant servers.

- **Pipelined two-phase commit:** CrossStitch introduces a pipelined two-phase commit protocol for ensuring the atomicity of a transaction. This protocol eliminates the requirement of a separate coordinator and it executes concurrently with the transaction. As a result, it provides CrossStitch with lower latency when completing a transaction.

- **Liveness and safety properties of CrossStitch's protocol:** We demonstrate that CrossStitch transactions, even in the presence of a single server failure, offer liveness. In other words, all CrossStitch transactions either commit or abort. Moreover, we show the safety of CrossStitch's pipelined two-phase commit protocol by demonstrating that it provides atomicity.

In our deployment, we find that CrossStitch is efficient and scalable. As expected, our experimental results demonstrate that the time to complete a transaction scales linearly with the chain length. Similarly, the number of concurrent threads also scales linearly with a transaction's completion time. We also find that by increasing the number of CrossStitch servers, CrossStitch is able support more clients. Our experiments show that a single client requires 14 ms to complete a transaction with five key accesses. We also find that CrossStitch provides significant latency improvements when a transaction accesses geographically distant servers. For a transaction with eight key accesses, we find that using

CrossStitch can provide up to a 39% improvement in latency over a traditional transaction processing system. Therefore, by offering an efficient and low-latency alternative to current transaction processing systems, CrossStitch has the potential to radically change how we perform transaction processing for web applications in cloud storage systems.

Chapter 2

Background and Related Work

In this chapter, we provide the background and motivation for CrossStitch. We further explore existing transaction processing systems and the mechanisms they use to provide transactional properties. Moreover, we look at current distributed storage systems and the motivation for their design and creation. We also describe the challenges that are present when providing transactions in modern cloud storage systems. Lastly, we explore current NoSQL transactional systems.

2.1 Transactions

As defined by Jim Gray [25], a transaction is a collection of operations on the physical and abstract application state that have the following ("ACID") properties:

- **Atomicity:** All of the transaction's operations and changes are applied, or none of the operations are applied.
- **Consistency:** A transaction does not violate any system invariants.
- **Isolation:** Transactions that execute concurrently appear to be executed sequentially.
- **Durability:** Once a transaction completes successfully or commits, the changes that it made persist in the event of a failure.

ACID properties for transactions are essential in ensuring application correctness. They ensure that all operations within a single transaction either complete or fail, that system invariants are not violated, and that the execution of a transaction appears to be in isolation from other transactions.

2.1.1 Motivation for Transactions

Transactions are desirable in many applications since they help ensure application correctness. A banking transaction is a canonical example that demonstrates how transactions can help to provide application correctness. Suppose one client wishes to transfer funds to another client. Atomicity is critical since one must ensure that the entirety of the transaction's operations are either executed or not executed. In particular, incorrect application behaviour would include debiting one account without crediting another account. Furthermore, a system invariant might include that a client's account balance cannot be negative. An application developer can use transactions to make certain that all system invariants are not violated. Furthermore, isolation is necessary to ensure that transactions accessing other accounts do not affect the correctness of our example. Lastly, once the transaction completes, durability is required to guarantee that changes to the clients' bank account persist in the event of failures.

In addition to the example banking transaction, many web applications can benefit from having transactions. As many web transactions are short and sequential in nature, a framework that provides low overhead when executing transactions is valuable. Furthermore, such a framework can contribute to ensuring application correctness. For example, ticket purchasing applications need to ensure that they do not double book patrons into the same seat and that the correct admission amount is transferred between the patron and seller. Moreover, many online games are deployed as web applications and many of these games involve purchasing in-game items or trading items among players. In order to achieve fair game play, transactions are necessary to ensure that no items are lost or inadvertently gained. Auction systems also require transactions as they need to ensure that the highest bid wins and that the correct amount is debited from the buyer and credited to the seller. We can imagine that any application that involves finances can benefit from having transactions, as transactions ensure that there is no loss of money, or unearned gain of money, for all parties involved. In addition to financial applications, any application that needs to ensure correctness of an operation can benefit from a transaction. For example, an internal messaging system, such as those found in social networking sites or business accounts, needs to ensure messages have been sent and received by the intended parties.

2.1.2 Existing Transaction Processing Systems

The focus of the cloud is to provide high scalability and reliability for its users. As a result, client data may be replicated at many locations or client data is sharded over various servers to provide reliability. Partitioning of data is necessary to ensure that the cloud is able to handle high volumes of data and client requests. Consequently, there is a need for distributed transactions in order to correctly operate upon client data. We define a distributed transaction to be a collection of operations that have ACID properties and are executed on more than one server. A common distributed transaction processing standard is the X/Open transaction processing model [4]. In this model, there are three primary components in a distributed transaction system: the application, the transaction manager, and resource managers. An application is responsible for defining a transaction's set of operations. An application may partition its work, such as retrieving data, into multiple client-server requests. However, transactions found in web applications typically consist of a single client that wishes to obtain information from various servers. As a result, we model an application as a single client that performs key accesses. Intuitively, a transaction program is implemented using the following steps:

- **Begin the transaction:** Upon beginning a transaction, the client contacts a transaction coordinator/manager to arrange timestamps and identifier.
- **Execute the transaction's operations:** The client contacts resource managers to access necessary resources.
- **Verify the transaction is successful:** Upon completion of all key accesses, a transaction manager or coordinator determines if all parties involved in the transaction can commit. If so, we commit the transaction.
- **Abort the transaction if necessary:** If one or more participants, such as resource managers, are unable to commit, the transaction manager or coordinator aborts the transaction.

When beginning a transaction, the application registers the transaction with the transaction manager in order to obtain a unique transaction identifier. In addition to providing transaction identifiers, a transaction manager is responsible for monitoring the transaction and determining if a transaction can commit or abort. After the transaction has begun, the application requests resources from the resource manager. A resource manager provides access to resources such as databases and is also responsible for ensuring that ACID properties are provided. Upon a resource request from an application, the resource manager

notifies the transaction manager that it is participating in the transaction. Consequently, the aforementioned resource manager participates in the decision of whether or not the transaction will commit or abort.

In order to ensure that transactions occur in isolated environments and that they do not conflict with each other, transactional systems need to employ concurrency control mechanisms. Many systems employ locking to ensure that only one transaction accesses a data object at any given time and to prevent other transactions from seeing uncommitted updates. In order to manage locks across distributed resources, a lock manager must be provided to resource managers in order to provide isolation. Although locks can be used to provide isolation, the transaction processing system's performance can significantly degrade if there is contention for resources. An alternative to locking is to use optimistic concurrency control. This model relies on the assumption that most transactions do not conflict with each other. However, in order to detect conflicts, an optimistic concurrency control scheme depends on a validation phase in order to determine if a transaction can commit. In the event of a conflict, the transaction is required to abort and the state of the system needs to be rolled back. An example of an optimistic concurrency control mechanism is multi-version timestamp ordering where many versions of a data object are maintained in order to provide a consistent view of data at a given time.

In addition to providing a lock manager, a transactional framework must also provide a log manager in order to ensure a consistent view of the system in the event of a failure. A log manager is responsible for recording all changes made by transactions; thus, it is able to reconstruct the system's state upon restarting, should a component of the system fail. The use of a log manager assumes that data is stored on one server; therefore, logs are required to reconstruct a server's state. However, in the case of NoSQL stores, reliability is provided through the existence of many replicas.

Once all of the resource managers have completed the application's requests, the transaction manager queries them to determine if the transaction is consistent and complete. In many systems, a multi-phase atomic commit protocol is employed to determine if there is a consensus among all participating resource managers. The transaction manager acts as the coordinator in two-phase commit and the resource managers are the participants.

2.1.3 Multi-phase Atomic Commit Protocols

Multi-phase atomic commit protocols are responsible for notifying all participants (resource managers) to commit or abort a transaction. If all participants in the transaction agree to commit a transaction, then the changes made by the transaction are applied locally on

the resources. Otherwise, if one or more participants decide to abort the transaction, then none of the transaction's changes are applied

Two-phase Commit

In order to determine if all participants are able to commit, many systems employ the two-phase commit protocol, which consists of a voting phase and a commit/abort phase. We describe the phases and the messages that are sent between the participants and coordinator using terminology similar to [36]. In the first phase, the coordinator will send a *vote request* to all participants in the transaction. Upon receipt of a *vote request*, a participant replies to the coordinator with a *vote commit* if it is able to commit. Otherwise, the participant sends a *vote abort* to the coordinator. After the voting phase, the coordinator is responsible for determining if all of the participants were able to reach a consensus. Unless every participant voted to commit the transaction, the coordinator sends a *global abort* message to the participants. If every participant voted to commit the transaction, then the coordinator sends a *global commit* message to the participants. If a participant receives a *global abort* message, then it ignores the transaction's changes and rolls back to its previous state. If a *global commit* message is received, then the participant applies the transaction's changes.

Although two-phase commit can atomically determine if all of the participants are able to reach a consensus, it has certain properties such that it is not suitable for web applications that demand low latency. Firstly, there is a significant amount of messaging overhead that is incurred between all participants and the coordinator, which in turn increases the transaction's latency.

Secondly, the coordinator in two-phase commit can also pose a performance bottleneck. As the coordinator blocks until all votes from participants are received (or there is a timeout), it becomes a source of contention if there are many entities (i.e., clients and servers) that wish to contact the coordinator. If there is a single transaction manager in a system, the transaction manager becomes a bottleneck when coordinating transactions between many clients and servers. Although a transaction may have its own coordinator, we find that many system implementations use a single entity for coordinating transactions. For example, systems such as ElasTras [18] use a single instance of ZooKeeper [26] in order to manage transactions' metadata and to provide coordination.

Lastly, a coordinator crash would render the participants unable to come to a consensus; thus, participants will have to wait for the coordinator to recover. In order to reduce the

need for participants to remain blocked until the coordinator recovers, three-phase commit was introduced.

Three-phase Commit

Similar to two-phase commit, three-phase commit [36] includes a voting phase where *vote request* messages are sent to all participants. However, unlike two-phase commit, a participant enters a state that indicates it has sent a *vote commit* message (call this the precommit state). Similarly, once the coordinator sends out a *global commit* message, it also enters the precommit state. If the coordinator fails, a participant that has not yet received a *vote request* can query other participants to see if they are in the precommit state. If so, then the transaction can be committed. Otherwise, the transaction is aborted. By maintaining an additional state and having participants query other participants in the case of a coordinator failure, three-phase commit trades off latency for resilience.

Paxos Consensus Protocol

In addition to two-phase commit and three-phase commit, Paxos [29] may also be used to determine if all participants are able to reach a consensus. Paxos is a quorum based consensus protocol that involves the use of proposers, acceptors, and learners. Proposers send prepare requests, which comprise a value and a request number, to acceptors. An acceptor may or may not accept the proposer's value depending on whether or not it has accepted a value which has a greater request number. If the proposer receives a majority of accept responses from the acceptors, then it issues accept messages to the the acceptors. As a result, acceptors receiving an accept message will accept the proposer's value. Once a value has been accepted, learners must determine that a proposal has been accepted by a majority of acceptors. This can be done by having an acceptor send the value to learners, or finding out from other learners the acceptor's response. In [24], Gray and Lamport show that two-phase commit is a special case of the Paxos commit algorithm.

2.2 Existing Cloud Storage Systems

Since the cloud is designed to be highly scalable and available, it has become the preferred platform for hosting web applications. The effectiveness of the cloud's underlying storage system has a significant impact on its overall performance. Most cloud storage systems

are designed as key-value stores where data is identified using a primary key. A key-value store is designed to allow users to retrieve and store information that is associated with a single key. In this section, we provide a description of key-value stores that are employed in industry.

2.2.1 Storage Models

Key-value stores generally partition data as a means to balance load across multiple servers. One method of partitioning data is consistent hashing [27] where each object is mapped to a single point on a line or ring. Subsequently, the line or ring is partitioned into multiple sections by some distance metric and each section is mapped to a server. An insertion or deletion of a single server does not require all keys to be remapped. Instead, a node insertion or deletion only affects keys in its neighbouring nodes. Systems such as Dynamo [21] (Amazon's key-value store) and Cassandra [28] (Facebook) use consistent hashing to distribute load. Other key-value stores, such as Bigtable [12] and Yahoo!'s PNUTS [13], use a directory service where different parts of the key-space are mapped to different servers. The use of a directory service will require load balancing to occur periodically. Many of these key-value stores allow for schemas which provide a structure for the data that is stored. For example, PNUTs organizes data into tables of records that have attributes and a key is required in order to index into a table. Similarly, Bigtable uses a distributed multi-dimensional sorted map where data is indexed using three dimensions: rows, columns, and timestamps.

2.2.2 Consistency Models

In order to achieve high availability and scalability, many cloud storage systems opt to omit strong consistency, which ensures that all replicas have the same state and see all updates. Although synchronous replication techniques provide strong consistency, they may not be appropriate for cloud storage systems. For example, read operations may need to be performed at the primary, which results in the primary server becoming a bottleneck. Moreover, writes need to be propagated synchronously to replicas, thereby causing the transaction to incur additional latency. In order to address the issues associated with synchronous replication mechanisms, a cloud storage system may employ asynchronous replication techniques where data is replicated in the background and use secondary nodes to serve requests, thereby providing eventual consistency. In other words, if no additional updates are made to an object in the database, all accesses would eventually obtain the latest version of the object [38].

For example, Amazon's Dynamo provides eventual consistency by asynchronously updating replicas, because it needs to maintain high availability for writes in order to service its shopping cart application. As a result, Dynamo uses vector clocks in order to assign versions to each read. In order to provide consistency, differing versions are resolved using vector clocks and quorums are used when executing get and put operations. In some cases, clients will read two versions and the client is responsible for reconciling the differing versions. Instead of eventual consistency, PNUTS provides per-record timeline consistency which guarantees that all updates are applied to all replicas of a given key in the same order. This ensures that a read or get operation will return a consistent view with respect to a record's timeline; however, this model is weaker than strong consistency. Finally, Cassandra [28] uses quorums to achieve consistency. Write requests are sent to a quorum of replicas that acknowledge the completion of writes. Depending on the client's requirements, a read request may be performed by one or many replicas.

2.2.3 Providing Reliability

The underlying systems for hosting web applications must be highly reliable as the inability to serve end-users, even for short periods of time, results in loss of revenue and customer trust. As a result, many systems employ multiple replicas for fault tolerance and reliability purposes. In Amazon's Dynamo, data is replicated over N nodes, where N is the number of copies for a key. This list of nodes, which is responsible for storing a particular key, is called a preference list. In the event of a node failure, Dynamo uses a sloppy quorum where read and write operations are performed on the first N healthy nodes, as opposed to the first N nodes on the preference list. Bigtable [12] uses Google File System [23] to store logs and data. Bigtable leverages the Google File System for maintaining multiple replicas of each file. Yahoo!'s PNUTS uses asynchronous replication in order to provide low latency. One copy of a record in PNUTS is designated as a master, and all updates are propagated to non-master replicas using a message broker service. Yahoo! chose this mechanism since their workloads exhibit write locality. In Cassandra, each data item is replicated at N hosts and a coordinator is in charge of data replication. In the event of a node failure, Cassandra relaxes quorum requirements for read and write operations.

2.3 Existing Distributed Transaction Systems

In this section, we survey existing storage systems that provide transactions. Firstly, we look at commercially available systems and we find that these systems utilize transaction

managers and coordinators to facilitate a transaction. Secondly, we describe key-value stores that provide transactional properties. Lastly, we outline works that provide transactions using various mechanisms including a shared memory model, timestamps, and other methods for providing transactional support.

2.3.1 Commercially Available Systems

There are various commercial databases that are able to provide distributed transactions. Such databases include Oracle [6], IBM's DB2 [3], and Microsoft's SQL Server [5]. All of these systems use a transaction coordinator to execute the transaction and two-phase commit to commit or abort the transaction.

Most commercially available databases follow the X/Open Distributed Transaction Processing Model, which we described in Section 2.1.2. IBM's DB2 follows the X/Open Distributed Transaction Processing Model in its transaction manager products. Microsoft's SQL Server does not inherently support transactions. Instead, transactions are provided through Microsoft Distributed Transaction Coordinator (DTC), which closely follows the X/Open Distributed Transaction Processing Model.

Oracle's system is a slight departure from the X/Open model. It comprises clients, database servers, global coordinators, local coordinators, and commit point sites. Suppose an application that wishes to access data on a remote server initiates a distributed transaction to access data that is stored on the database server. The server on which the application initiates the transaction is called the global coordinator. A global coordinator is responsible for sending the distributed transaction's SQL statements and executing remote procedure calls. Furthermore, a global coordinator may also be a local coordinator, since a local coordinator is defined as a node that must reference data on other nodes in order to complete its part of the distributed transaction. Once the global coordinator has completed the transaction, it notifies the commit point site, which is a node that commits or rolls back the transaction as instructed, to commit a transaction. In order to commit, Oracle's system employs two-phase commits.

The use of a coordinator in Oracle, DB2, and Microsoft's SQL server may result in greater latency as applications may need to continuously communicate with transaction and resource managers before proceeding with the transaction. The use of a coordinator introduces a single point of failure, thereby decreasing robustness. Furthermore, a centralized coordinator may hinder the scalability of a system as it introduces a performance bottleneck.

2.3.2 Transactional Key-Value Stores

In addition to commercially available database systems, other works such as G-Store [20] and Megastore [9] provide transactions by grouping and/or partitioning keys. These works restrict transactions to certain partitions and key groups.

G-Store provides transactions on multi-key accesses by using a key group abstraction that defines a relationship between a group of keys. G-Store's key grouping protocol uses a key group to transfer ownership of all keys in a group to a single node. Within a key group, G-store assigns a leader key, which is used as part of the identity of the key group, and the remaining keys are follower keys. The node that owns the leader key is assigned ownership of the key group and can guarantee consistent access to keys in the key group without distributed synchronization. Consequently, the size of a key group is limited to the number of keys that can be hosted on a single node. At any given time, a key can be a part of a single key group; however, a key may be a part of multiple groups over its lifetime. As a result, a transaction's key requests are limited to keys that are stored in a single node. G-Store also employs write-ahead logging to achieve durability. Using this key group abstraction, G-Store offers transactional multi-key access guarantees over a non-overlapping group of keys within a key-value store.

Similar to G-Store, Megastore [9] provides ACID semantics within fine-grained partitions of data. Megastore partitions the data-store and replicates each partition separately; thus, Megastore is able to have transactional guarantees within a single partition; however, it can only offer limited consistency guarantees across the partitions. In order to scale throughput and localize outages, Megastore partitions data into a collection of entity groups that are synchronously replicated over a wide area. Entities within an entity group are mutated with ACID transactions, and the commit record is replicated via Paxos. Operations across entity groups rely on two-phase commits, which are relatively expensive.

Therefore, by partitioning data, systems such as G-Store and Megastore are able to provide transactions over a well-defined partition. However, both systems require knowledge of all key accesses in order to create key groups or entity groups so that transactions may be executed efficiently.

2.3.3 Using Shared Memory to Provide Transactional Support

Instead of partitioning data into fine-grained modules so that transactions may occur, other work such as Sinfonia [8] uses a shared memory model. Sinfonia removes message passing protocols between distributed systems and simply allows developers to manipulate data

structures and allow applications to share data. As a result, Sinfonia presents lightweight minitransactions, which are primitives that applications can use to access and modify data. A minitransaction consists of a compare items set, a read items set, and a write items set. During its execution, a minitransaction checks data as indicated by the compare items set, and if all comparisons succeed, the minitransaction returns the memory locations of the read items and write items. The minitransaction is piggybacked onto the first phase of the two-phase commit protocol. The Sinfonia infrastructure consists of application nodes, where applications are run, and memory nodes, which hold application data. Sinfonia primarily targets infrastructure applications (e.g., cluster file systems, lock managers, and group communication servers) instead of typical applications that are found in the cloud.

2.3.4 Snapshot Isolation and Timestamp-Related Mechanisms

Systems such as Spanner and Walter use timestamp-related mechanisms to provide transactional support. Spanner [15] is Google's multi-version database that is globally-distributed, synchronously-replicated, and designed for long lived transactions. Spanner stores data in tablets, where a tablet is a collection of mappings that map a key and timestamp to a string. Spanner assigns global commit timestamps which are used to reflect serialization order. Thus, it is able to provide externally consistent reads and writes and globally-consistent reads across the database at a given timestamp. Spanner supports read-write transactions, read-only transactions, and snapshot reads. Write transactions in Spanner are buffered at the client until the transaction commits, and once a client has completed all reads and buffered writes, it executes the two-phase commit protocol. Currently Spanner is used to serve Google's advertising back-end.

Walter [35] is a key-value store that supports transactions and asynchronously replicates data across distant sites. Walter utilizes a new property called parallel snapshot isolation (PSI) to replicate data asynchronously while providing strong guarantees within each site. Walter uses preferred sites to implement PSI and prevent write-write conflicts. Parallel snapshot isolation enforces causal ordering of transactions and allows different commit orderings at different sites. Causal ordering is defined as follows: if a transaction T_2 reads information from T_1, then T_1 is ordered before T_2 at every site. Walter utilizes preferred sites where each object is assigned a preferred site and writes to the object at the preferred site can be committed without checking other sites for write conflicts.

2.3.5 Transaction Coordination Systems

Systems such as Cloud TPS [39] and Granola [17] utilize a transaction coordination system in order to execute transactions. In Cloud TPS, the authors observe that a centralized transaction manager would become a bottleneck, as it must execute all incoming transactions, and it would run out of storage space since a transaction manager needs to keep a local copy of all data accesses. The intuition behind Cloud TPS is that it splits the centralized transaction manager into a number of local transaction managers. Similar to CrossStitch, Cloud TPS aims to provide transactions for web applications. Cloud TPS implements a transaction as a group of subtransactions. If subtransactions, within a single transaction, have data conflicts, then the subtransactions are executed sequentially. Cloud TPS requires that the application provide the primary keys of all data items that a single transaction wishes to access.

Granola is another transaction coordination infrastructure that specifically supports independent transactions, which are transactions that execute atomically across a set of participants and do not require locking since they do not contend with other transactions. Granola serializes these transactions using its own timestamp-based coordination mechanism in order to provide lower latency and higher throughput. There is a single round of communication between the client and a set of repositories (participants). These one-round transactions do not allow for interaction with the client (which means that client cannot execute multiple sub-statements or queries). One-round transactions must also execute to completion at each repository (participant) with no communication to other repositories. Granola's repositories execute transactions and communicate with one another to coordinate the transaction.

2.3.6 Separating Transaction Processing from the Database

There have been many works which separate transactional processing from the data storage layer. ElasTraS [19] is a light-weight data store that supports a subset of operations that are supported by traditional database systems. ElasTraS uses a key-value store for data storage and uses a two-level hierarchy of transaction managers in order to provide transactional guarantees. ElasTraS allows data to be partitioned statically or dynamically. If an ElasTraS data store is partitioned statically, it can provide ACID transactional guarantees for transactions limited to a single partition. If an ElasTras data store is partitioned dynamically, it only supports minitransactions, which have restricted transactional semantics, ensuring recovery but not global synchronization. Similar to other decoupled architectures, ElasTraS partitions functionality into various layers: the data storage layer,

owning transaction managers, and higher level transaction managers. The data storage layer is responsible for replication and fault tolerance. The owning transaction managers are responsible for the execution of transactions on the partitions of databases, concurrency control, and recovery functionality. The higher level transaction managers absorb all read-only transactions and cache a subset of the database for read-only purposes.

Other works such as Lomet et al. [31] propose providing transactions by factoring a cloud storage engine into two layers: a transactional component and a data component. The transactional component works at a logical level and a data component is responsible for the physical storage structure. The transactional component is responsible for the locking, concurrency control, and logging, for the purposes of durability. Only the data component has knowledge of the pages; thus, the transactional component must invoke operations for the data component. The transactional component is responsible for ensuring isolation, transactional atomicity, and logging. Similarly, the data component is responsible for providing atomic operations on the data, maintaining indexes and storage structures (e.g., if data is stored in a structure such as a B-tree) , and providing cache management. The notion of unbundling the transactional component and data component is also found in Deuteronomy [30]. Deuteronomy supports ACID transactions by decomposing the database into a transactional component and a data component. A transactional component is responsible for managing transactions, concurrency control, and recovery. A data component maintains a cache and accesses data. When a client wishes to execute a transaction, the transactional component performs all operations that are necessary for the transaction (e.g., logging/locking), and routes data update operations to the correct data component.

Another transactional system that uses a modularized approach is Calvin [37], which provides transaction scheduling and a data replication layer. Calvin is designed to run alongside a non-transactional storage system. In order to reduce contention, Calvin servers determine how to handle a transaction before acquiring locks and beginning to execute a transaction. Once all Calvin servers have determined how to handle a transaction, the transaction must be executed to completion. Thus, Calvin consists of a sequencing layer that intercepts transactional inputs and places them into a global transactional input pattern. During every epoch, which refers to a 10 ms time frame where a server collects transaction requests from clients, Calvin's sequencer collects and replicates the transactional requests. Afterwards, a message is sent to the scheduler on every partition for the purposes of replication. Calvin's scheduling layer is then responsible for the execution of the transaction and it uses a deterministic locking scheme to guarantee equivalence to the serial order that is specified using the sequencing layer. The storage layer handles the physical data layout. Calvin does not support transactions that must perform reads in

order to determine their full read/write set. This is due to Calvin's deterministic locking protocol which requires advance knowledge of all transactions' read/write sets.

Chapter 3

CrossStitch: An Efficient Transaction Processing Framework for Web Applications

In this chapter, we provide an overview of the CrossStitch transaction processing system. We describe the structure of a CrossStitch transaction and its operations. We also illustrate the method for implementing a CrossStitch transaction. We then explain CrossStitch's messaging protocol and the servers' messaging pattern. Lastly, we outline the CrossStitch architecture and detail its design.

3.1 Basics of a CrossStitch Transaction

CrossStitch is a transaction processing framework that adds transactional support for key-value stores. CrossStitch clients submit their transactions to CrossStitch servers which are responsible for maintaining the transaction's changes until the transaction commits. Moreover, CrossStitch ensures the atomicity of a transaction and that a transaction aborts if any isolation or consistency constraints are violated.

3.1.1 Implementing a Transaction

As discussed in Section 1.2, CrossStitch transactions are structured as a series of components, where each components consists of a single key access (i.e., get, put and delete) and

additional computation. We refer to the series of components as a transaction chain. Intuitively, a component is analogous to a link in a chain. A CrossStitch transaction proceeds to the next component when a key access is executed; therefore, the transaction's next component is specified each time one of these operations is called. The terminating component of a transaction chain is identified by the lack of a key access. A detailed example of how to implement a transaction is provided in Section 3.2.

Each key operation produces a value that is used as a parameter to the subsequent component in the transaction chain. A get operation retrieves the value in the key-value store and this value is passed as a parameter into the next component in the transaction chain. Delete and put operations produce an empty value that is generally unused by the subsequent component. The terminating component's return value is the final value that is returned by the transaction; this value is also sent to the client.

In our CrossStitch implementation, a transaction is implemented as a class that inherits from a base class that implements the transaction's operations. Each method in the transaction class represents a single transaction component that comprises a key access and some computation. We use Python in our CrossStitch implementation.

3.1.2 CrossStitch Timestamps

CrossStitch assigns a client-specified timestamp to each transaction in order to determine ordering. CrossStitch uses a variant of multi-version timestamp ordering to provide serializability; thus, a transaction's timestamp is used to determine which version of data is retrieved and it serves as metadata when writing an item to the key-value store. All CrossStitch servers and clients need to be loosely time synchronized in order to allow for timestamp comparison. The system time of servers determines the ordering of transactions and it determines if a transaction is too stale and needs to be rejected. Therefore, using clients and servers with vastly different clocks significantly impacts transaction ordering and results in a greater abort rate. However, the differences between client and server clocks do not affect correctness as the ordering of all transactions is reflected on all servers.

Since multiple transactions can begin at the same time, a CrossStitch timestamp also combines a client identifier and a transaction identifier to ensure uniqueness. Client identifiers uniquely identify each client in the system and transaction identifiers are unique within each client. Therefore, a transaction as a whole is uniquely identified by the time, client identifier, and transaction identifier. CrossStitch maintains additional state in order to ensure that the pipelined two-phase commit protocol, as we describe in Section 3.2.2,

proceeds correctly. As mentioned previously, a CrossStitch transaction consists of multiple, sequential components that form a chain and more than one component may be executed on a single server. Therefore, in order to uniquely identify each component in the transaction chain, we also use a chain identifier, which refers to the n^{th} component of the transaction. Therefore, a CrossStitch timestamp of an individual component (i.e., key access and computation) comprises system time, a client identifier, a transaction identifier and a chain identifier.

3.1.3 CrossStitch Concurrency Control

CrossStitch uses multi-version timestamp ordering that provides optimistic concurrency control. Every version of an item in CrossStitch's datastore maintains a read timestamp and a write timestamp. The write timestamp indicates when the object was created. The read timestamp indicates the most recent time that the object was accessed. Both timestamps are used to determine if a transaction needs to abort.

CrossStitch maintains multiple versions of an item in order to provide concurrency control. When executing a key operation such as a get, put, or delete, the transaction's timestamp is compared with the timestamp of items found in the underlying data store in order to determine the version of the item that is retrieved. This comparison is done immediately when a transaction arrives and is executed at the server since a CrossStitch component consists of a key access followed by computation. When a transaction retrieves an object that is associated with a given key, it invokes the get operation. CrossStitch returns the version (call this v) of the object that satisfies the following two conditions:

- The write timestamp of the committed returned version (v) is earlier than the transaction's timestamp.
- There does not exist another committed version of the object that has a write timestamp more recent than v and earlier than the transaction.

3.1.4 Timestamp Related Abort Cases

As the CrossStitch transaction chain progresses, all pending get, put, and delete operations are maintained by the server executing the key request. Upon the receipt of a commit message, the server will write the pending operations into the underlying datastore. As CrossStitch utilizes multi-version timestamp ordering, it maintains multiple versions of an

object in its datastore. Each version of each object x has an associated read timestamp (*rts*) and a write timestamp (*wts*). An object's *rts* and *wts* are unaffected by pending operations. However, once a pending read or write is committed, the object 's *rts* or *wts* is updated accordingly. In order to ensure that a transaction's view of an object is consistent, CrossStitch's transaction processing layer verifies that a write operation does not invalidate a read operation. If a transaction violates any of the timestamp conditions as described in this section, then it is aborted. A server immediately determines whether or not a key operation aborts once the transaction arrives. The component's execution comprises a key access, followed by computation. The key access and the check for timestamp violations are performed before any additional computation. If the key access results in a timestamp violation, then the transaction is aborted.

For this section, let $rts(x)$ denote the read timestamp of object x, and let $wts(x)$ denote the write timestamp of object x, where x is a version of an item. We use the terms *previous pending read (PPR)* and *previous pending write (PPW)* to denote the most recent (i.e., highest timestamp value) pending get and put operations that are before the transaction's timestamp. Similarly, *previous committed read (PCR)* and *previous committed write (PCW)* refer to most recent committed get and put operations that are before the executing transaction's timestamp. Suppose a transaction wishes to access key x. CrossStitch aborts for the following read-write and write-write conflicts:

- **Reading an uncommitted write:** The transaction aborts when a get operation attempts to read a version of an item that has not been committed. If $ts_{PCW} < ts_{PPW} < ts_{transaction}$, then the transaction may read a value that is later aborted. As a result, we abort the transaction. Our multi-version timestamp ordering concurrency control implementation aborts if a read would access an uncommitted write. This allows for a simpler implementation at the expense of an increased abort rate. This is a simplification of the traditional multi-version timestamp ordering mechanism that blocks until the transaction containing the uncommitted write operation completes.

- **Invalidating a read:** Let $ts(T_i)$ denote our transaction's timestamp, and T_i wishes to write to the key x. Let x_{prev} be a previous committed version of the object that is referenced by x. If x does not exist in the datastore (i.e., it has not been written to), then $rts(x_{prev}) = -\infty$. The transaction aborts if $rts(x_{prev})$ is more recent than $ts(T_i)$ since T_i will invalidate a read operation that is performed by another transaction. Let NPR (next pending read) denote a pending read operation such that $ts(T_i) < ts(NPR)$. If there does not exist a committed write (CW) such that $ts(T_i) \leq ts(CW) \leq ts(NPR)$, then T_i aborts since the read operation NPR would become invalidated.

CrossStitch aborts a transaction if the transaction's timestamp is older than the oldest version of the requested item. It maintains a limited number of versions in its datastore to bound storage requirements and keeps the most recent version of items in its datastore.

In addition to our variation of the multi-version timestamp ordering mechanism (i.e., reading an uncommitted write causes an abort), we have already begun exploring a new variant of this optimistic concurrency control mechanism in order to lower CrossStitch's abort rate. In order to prevent reads from aborting, a read operation may need to block and wait for a pending write to be committed. However, this may not be appropriate for web transactions since they are sensitive to latency. As a result, our new variant of multi-version timestamp ordering allows a transaction to continue by performing the read operation on an uncommitted write. However, a server does not indicate that it is ready to commit until the pending write operation has been committed. The trade-off is that if the transaction for the pending write operation is aborted, it may cause a cascading abort for the transaction that contains the read operation. Preliminary results demonstrate that this variant of multi-version timestamp ordering results in a lower abort rate for high contention workloads. This more complex variant is not described in this thesis and is left as future work.

3.1.5 Non-timestamp Related Abort Cases

In addition to the abort cases described in the previous section, a transaction may also be aborted if it violates certain system conditions (e.g., incorrect implementation or the transaction chain forms a loop) or if the application (e.g., end user) wishes to abort.

System Conditions Violated

A client can create an infinite loop in the transaction by creating component transitions that occur in a cycle. In the event of an infinite loop, the transaction does not complete, and it would consume resources, making them unavailable to other transactions. As a result, a transaction aborts after it has reached a predetermined number of hops or operations. This number is significantly greater than our expected web transaction so that transactions do not unnecessarily abort.

Users Requesting Termination

Users may also request the transaction to be aborted by calling the abort function in the transaction's underlying base class. A user may want to do this when its logical constraints are violated. For example, in a bank transaction, a user's account may not be negative after a debit operation occurs. Suppose a client's balance is retrieved, and it does not have sufficient funds to make the requested withdrawal. The client needs to be able to abort the transaction.

3.1.6 CrossStitch's Supported Operations

CrossStitch supports three primitive operations: get, put, and delete. Upon the arrival of a transaction, the server immediately executes the get, put, or delete operation that is associated with its assigned component in the transaction chain. When the key operation is executed, CrossStitch determines if the transaction needs to be aborted due to timestamp violations, which were described in Section 3.1.4. After executing the key operation, the server maintains, or buffers, the operation and the operation's corresponding timestamp until the operation's transaction commits. Once a transaction commits, all of its operations (i.e., gets, puts, deletes) are applied to the datastore with the corresponding timestamp.

An end-user invokes a get operation by calling the function below.

```
transaction.get(next_component, parameters, key)
```

next_component indicates the next user-defined component in the transaction chain that will be run by the server once the get operation completes. A user-defined component is a method that implements a part of the transaction's function. The client specifies a finite-sized list called *parameters*, which is used as input for the next component in the transaction chain. Lastly, *key* refers to the key that the end-user desires to access.

When executing a put operation, CrossStitch adds an object into the datastore that is indexed using the provided key. The write timestamp indicates an object's version. A put operation is called by the end-user as follows:

```
transaction.put(next_component, parameters, key, value)
```

The *next_component*, *parameters*, and *key* parameters are as described for the get operation. The put operation also specifies *value*, which is the object that is written into the datastore and is indexed using *key*.

A delete operation is implemented using the put operation, which was described previously. However, instead of specifying a value, CrossStitch writes a pre-defined empty object, which is used to indicate that a key's value has been deleted, into the datastore using the transaction's timestamp. The delete operation's parameters are identical to the parameters of the get operation. A delete operation is called by the end-user as follows:

```
transaction.delete(next_component, parameters, key)
```

3.2 An Example Transaction

In this section, we provide an example transaction to explain the structure of a CrossStitch transaction, illustrate the messages that are sent between CrossStitch servers and demonstrate how CrossStitch transactions are executed. Suppose we want to implement a transaction that accesses key x and increments its value by one. The transaction also adds the value of y to the value of x before it was incremented in order to construct its return value. Intuitively, a transaction would be implemented in the following manner:

```
BEGIN_TRANSACTION
value = get(x)
put(x, value+1)
second_value = get(y)
return (value + second_value)
END_TRANSACTION
```

Figure 3.1: An Example Transaction

In the CrossStitch framework, the application developer would decompose the transaction in Figure 3.1 into a series of components that form the transaction chain, where each component, other than the final component, comprises a single key access and computation. The final component in the transaction chain does not perform an additional key access and provides a return value back to the client. An example CrossStitch transaction is shown in Figure 3.2. We also call the transaction that is specified in Figure 3.2 as *example.py*.

The transaction's execution shifts from one component to another when it accesses a key through a get, put, or delete operation. Each key access requires the name of the next

```
import transaction
class basic_add(transaction.transaction):
   def start_component(self):
      return self.get("comp_2",[], "x")


   def comp_2(self, val):
      new_val = val + 1
      return self.put ("comp_3", [val], "x", new_val)



   def comp_3(self, val, p1):
      return self.get("comp_done", [p1], "y")



   def comp_done(self, val, p1):
      return val + p1
```

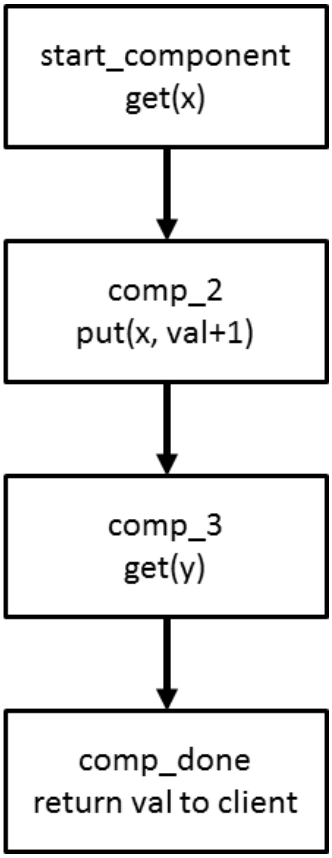

Figure 3.2: An Example Transaction's Implementation.

component, the value returned by the previous key access, and a list of parameters as its input. The number of parameters in the subsequent component must match the input parameter list's length.

We also note that *start_component* is not referenced by any other state. In our implementation of CrossStitch, the transaction's initial component is called the *start_component*. The client begins the transaction by executing the *start_component*, and the transaction runs on the client until the *start_component* makes a key request.

3.2.1 Launching and Running a Transaction

```
tclient.client(host_name = None,
               port_num = None,
               cid = None)
```

Figure 3.3: Instantiating a CrossStitch Client

The application developer must create a CrossStitch client in order to execute a transaction. Figure 3.3 demonstrates how to instantiate a CrossStitch client. Once the client has been instantiated, it calls *start_transaction* to launch the transaction. *start_transaction* requires the following parameters:

- **Timestamp:** This is the system time that is used to construct the transaction's timestamp.
- **Implementation file:** This file contains the transaction's code. An example of this is Figure 3.2. We call the code presented in Figure 3.2 *example.py*.
- **transaction_ class (transaction name):** This is the name of the transaction class that is to be executed. In the case for Figure 3.2, the client specifies *basic_add*.
- **start_ comp_ params (user-defined parameters):** This is a list of parameters that the end-user wishes to pass to the first component in the transaction chain.

The *start_transaction* function is defined in Figure 3.4:

```
client.start_transaction(ts,
                         transaction_file,
                         transaction_class,
                         start_comp_params=None)
```

Figure 3.4: API Call to Start a Transaction

Figure 3.5 demonstrates how an end-user would create a client and launch a transaction. The end-user launches the *basic_add* transaction from *example.py*. It uses the system time and does not pass in any parameters to *start_transaction*.

```
import time
import tclient

client_one = tclient.client()
return_value = client_one.start_transaction(
                                            time.time(),
                                            "example.py",
                                            "basic_add",
                                            [])
```

Figure 3.5: Creating and Launching a Transaction

3.2.2 Basic Messaging Chain

In order to provide low latency, CrossStitch minimizes the number of messages sent between the client and servers by executing the transaction's computation on the servers and by forwarding the transaction to servers that are responsible for the key accesses. Moreover, CrossStitch performs a pipelined two-phase commit protocol while the transaction is being executed on the servers. As a result, CrossStitch can significantly reduce latency and completion time in the case where client to data-center latency is high. Figure 3.6 shows the messages that are sent among all participants for our example transaction.

Intuitively, throughout the execution of the transaction, CrossStitch's pipelined two-phase commit protocol transfers the role of the traditional transaction coordinator to the server that is executing the key access. When a transaction arrives at a server, it is executed in order to determine the next key access. The next key in the transaction is sent back to the previous server in the form of an acknowledgement message. By having knowledge of subsequent keys in the transaction chain, a server may query another server that is later in the transaction chain for the purposes of failure handling and recovery. In addition to the acknowledgement message, a server sends a precommit message to the next node if it is ready to commit (i.e., has voted commit in the traditional two-phase commit protocol). In effect, each server enters the *vote commit* state as the transaction progresses. Once the last server in the transaction chain completes executing its assigned component, it notifies its replica, which is another datastore, that it is ready to commit. In this context, the replica is responsible for recording the successful completion of the transaction and is not a data replica. We later discuss its use for fault tolerance in Chapter 5. The last server then proceeds to send commit messages to all participating servers and the result back to the client.

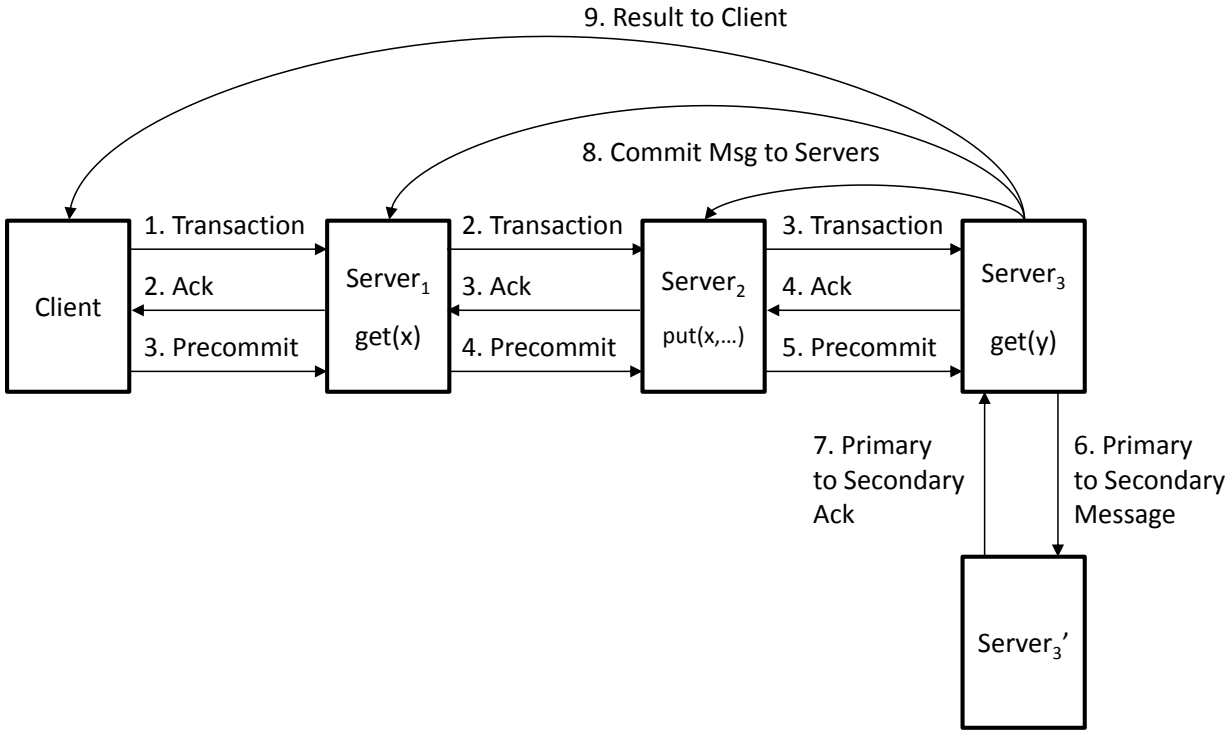

Figure 3.6: CrossStitch's Messaging Pattern

In Figure 3.6, the client begins by executing *start_component* until it reaches a key request. The client determines that $Server_1$ is responsible for hosting key x, and the transaction is forwarded to $Server_1$. Upon receiving the transaction message, as indicated by the label *"1. Transaction"*, $Server_1$ executes *comp_2* until it reaches the put request. Similar to the client, $Server_1$ determines the server that is responsible for the put(x) key access, which is labelled as $Server_2$ in Figure 3.6. When $Server_1$ sends the transaction to $Server_2$, it also sends an acknowledgement message to the client. The acknowledgement message notifies the client that $Server_1$ was able to successfully execute its assigned component in the transaction chain. The client then replies with a precommit message, which notifies $Server_1$ that it has received the acknowledgement message and is now waiting for the final result.

We note that $Server_1$ and $Server_2$ are the same physical server as they both access the same key. However, we name these hops $Server_1$ and $Server_2$ to denote the hops in the transaction chain. Generally, two consecutive hops in a transaction chain may be found on the same server. The CrossStitch framework handles this case by having servers send

messages to themselves.

After $Server_1$ has sent the transaction to $Server_2$, $Server_1$ waits for an acknowledgement message from $Server_2$ and a precommit message from the client. $Server_1$ can shift to the precommit state and send a precommit message to $Server_2$ once it has received these messages. By doing so, $Server_1$ indicates that it is able to commit and is waiting for a commit or an abort message. Therefore, the precommit message is analogous to the *vote commit* message in two-phase commit. As the transaction progresses, $Server_2$ executes *comp_3*, and $Server_3$ executes *comp_done*. The messaging between $Server_2$ and $Server_3$ is identical to the messaging between $Server_1$ and $Server_2$. After successfully executing the transaction, $Server_3$ sends a primary to secondary message to the transaction's replica, $Server'_3$. As $Server'_3$ is now aware that the transaction has completed, in the event that $Server_3$ fails, $Server_2$ may query $Server'_3$ in order to determine if the transaction is successful. $Server'_3$ replies with a primary to secondary acknowledgement message to notify the primary server that it has a record of the transaction. Once $Server_3$ receives the primary to secondary acknowledgement message, the transaction is considered to be committed and $Server_3$ returns the value to the client and notifies all servers in the transaction chain to commit.

3.2.3 CrossStitch Messages

CrossStitch employs a number of messages to execute its pipelined two-phase commit protocol. Intuitively, these messages are used to forward the transaction from server to server, to acknowledge the receipt of a transaction, and to confirm that a server is ready to commit. In this section, we describe the messages that are sent between CrossStitch servers and clients.

- **Transaction Message**: In addition to the client identification number, transaction identification number, and timestamp, this type of message also contains transaction code and information, which includes the name of the next component in the transaction chain, parameters, and the result of the previous component.
- **Acknowledgement Message**: Once a server receives a transaction message, it executes its assigned component to determine the next transaction operation or the returned value. Afterwards, the server sends an acknowledgement message to the sender of the transaction message. This message may contain the server that is the next hop for fault tolerance purposes as described in Chapter 5.

- **Precommit Message**: A precommit message is used by a client or server to notify a subsequent server that it is ready to commit. It is sent once a server has received all of its expected messages or if a client has received an acknowledgement message. These messages include the transaction, the acknowledgement message, and the precommit message from the previous node in the transaction chain. A precommit message has the same functionality as a vote commit message in two-phase commit.

- **Primary to Secondary Message:** The primary to secondary message is sent from the transaction's final server to the final server's replica. This message is used to notify the replica that the transaction has completed and is able to commit. By having knowledge of the transaction in the replica, CrossStitch is able to tolerate up to one server failure.

- **Primary to Secondary Acknowledgement Message:** The primary to secondary acknowledgement message is sent from the transaction's final server's replica to the primary. The purpose of this message is to notify the primary server that the transaction has been recorded.

- **Commit Message**: When a transaction has completed successfully, the last server in the CrossStitch transaction chain sends a commit message to all servers that participated in executing the transaction. Once a server receives a commit message, it commits the transaction by applying the pending operation to the datastore.

- **Abort Message**: If a transaction aborts (e.g., due to conflicting write operations), then the server that is responsible for the abort sends abort messages to all servers that have participated in executing the transaction. Once a server receives an abort message, it removes the transaction's pending operation and transitions to the abort state.

- **Result Message**: This message is sent by the last server in the transaction chain (also called an end server) to the client and it contains the result of the transaction. The server sends this message to the client once it has committed the transaction.

3.3 Managing the Transaction on the Server

In order to implement CrossStitch's pipelined two-phase commit protocol, a significant amount of state must be maintained for each operation that occurs. In this section, we explore the functionality of the server and we describe our server implementation. We

illustrate the state machines that CrossStitch servers maintain. Lastly, we describe how CrossStitch interacts with the underlying datastore.

3.3.1 Server Configuration

In our implementation of CrossStitch, all servers are connected to each other, forming an n-complete graph. CrossStitch servers maintain persistent TCP connections with each other in order to avoid incurring additional latency that is the result of performing a TCP handshake when establishing a connection. By establishing all connections beforehand, CrossStitch servers can easily communicate with each other as the transaction progresses. For similar reasons, a client creates a connection to each CrossStitch server so that connections to not have to be established and closed each time a client executes a transaction. Although this configuration requires n^2 connections to be made, it may limit scalability. However, n is sufficiently small in practice such that it is feasible to establish all connections beforehand. For example, Google's Bigtable [12] is built for thousands of servers.

The underlying storage system is responsible for determining the location of the keys. In our implementation of CrossStitch, we use a simple storage system that assigns keys to servers using the modulus of a key's hash value. For example, if the servers were indexed 0 through $n - 1$, then the server responsible for hosting k is k's hash value mod n.

3.3.2 Maintaining State Machines

Until a transaction is committed or aborted, a server maintains information regarding the transaction in case the transaction aborts later in the chain or conflicting writes occur. In order to maintain this information, each server maintains state for all of its assigned components in the transaction chain and a list of pending operations. The pending operations are applied when the server receives a commit message, or they are removed when the server receives an abort message.

In particular, a server needs to determine the messages that it is expecting for a key access and computation (e.g., an acknowledgement from the next server, a precommit from the previous server, or a notify message from the previous server's replica). By maintaining a component's state, a server is able to determine when it can send the appropriate messages (e.g., a precommit message to the next server). Keeping track of the component's state is imperative for ensuring CrossStitch's correctness.

- **Start State:** When a new transaction arrives at a server, it begins at the start state once it arrives at its assigned server. Once the server completes executing its assigned component in the transaction chain, the component's state transitions from the start state to the *Wait for Messages State*.

- **Wait for Messages State:** In this state, the server waits for all of the component's expected messages. For components in the transaction chain that are not the last hop in the chain, this includes a precommit message from the previous server and an acknowledgement message from the next server. Once all of the expected messages are received, the server sends a precommit message to the next server, and the state of the component changes to the precommit state. Components that are the last hop in the transaction chain only wait for the precommit message from the previous server before transitioning to the Primary to Secondary Message Sent State.

- **Precommit State:** A component in the transaction chain enters this state if and only if it has been successfully executed and if its server has received an acknowledgement from the next server in the transaction chain and a precommit message from the previous server in the transaction chain. From this state, the server cannot abort the transaction. This state aborts if and only if it receives an abort message from a subsequent state in the transaction chain. The component may enter the abort state or commit state from the precommit state.

- **Primary to Secondary Message Sent State:** The end server in the transaction chain enters this state once it has received a precommit message from the previous server. Upon entering this state, the server responsible for the final component sends a primary to secondary commit request message to its replica, which is another datastore. The primary to secondary commit request message notifies the replica that the transaction has successfully completed. Once the replica replies with a primary to secondary commit acknowledgement, the final component's state transitions to the commit state.

- **Commit State:** When a component is in the precommit state and it receives a commit message, its state moves to the commit state. This state indicates the transaction has completed successfully.

- **Abort State:** When a component is in the precommit or the wait for messages state and it receives an abort message, the component moves to the abort state. A component can also move to the abort state from the start state when it executes the transaction, and the transaction aborts due to a key conflict.

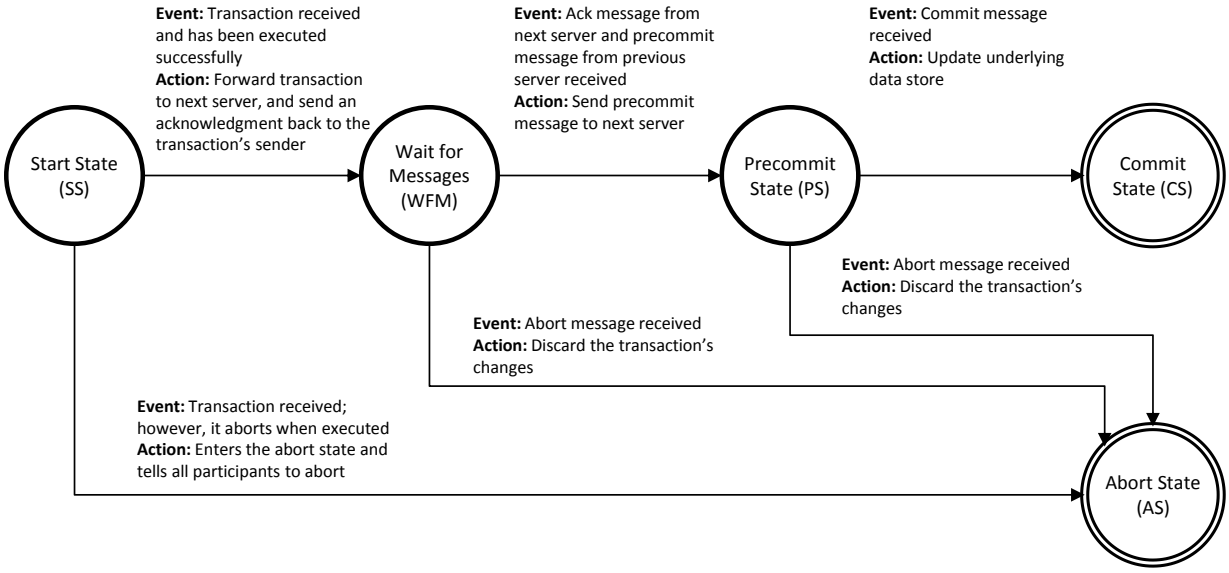

Figure 3.7: Intermediate Server's State Machine

In this section, we describe how a server transitions between various states for its assigned key access and computation. We call the last server in the transaction chain an end server, and its state machine is depicted in Figure 3.8. All other servers in the transaction chain are called intermediate servers. The state machine of an intermediate server is shown in Figure 3.7. Upon the receipt of a new component, CrossStitch constructs a new *start_state* object on the server. The *start_state* object is indexed using the state's timestamp, which consists of system time, transaction identifier, client identifier, and chain identifier.

Upon receipt of subsequent messages that have the *start_state* object's identifier, the server entry for the transaction is updated with a new state object that reflects the messages that have been received. For example, when a transaction's key access is received at a server, the entry in the state machine shifts from the *Start State* to the *Wait for all Messages State*. While in the *Wait for all Messages State*, the server waits for an acknowledgement message from the next server (to ensure that the transaction has been received), and a precommit message from the previous server. Once the server has received all messages that it is expecting, it enters the precommit state (as shown in Figure 3.7. If the server's assigned component is the last hop in the transaction chain, the server notifies its replica that it is ready to commit (shown in Figure 3.8). The component's state enters the commit state if one of the following conditions holds:

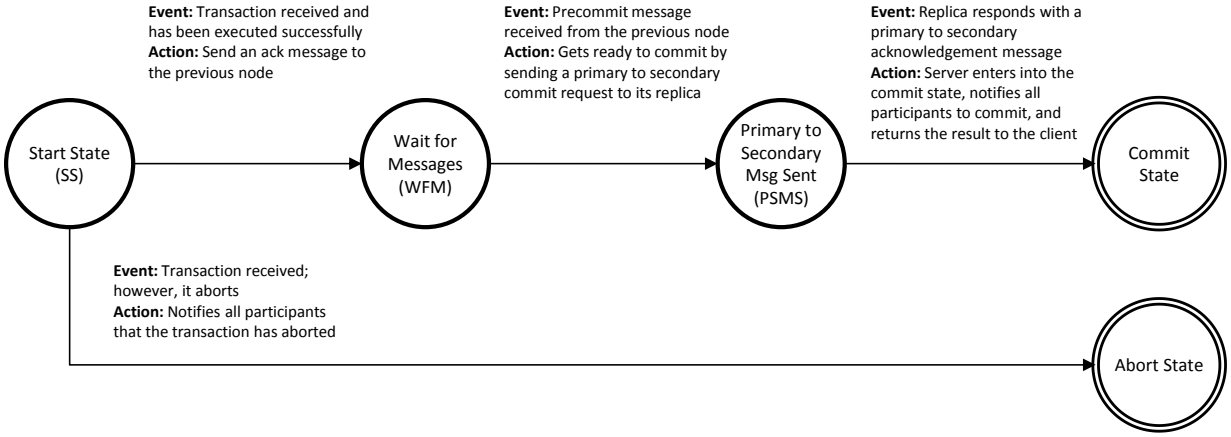

Figure 3.8: End Server's State Machine

- It is in the precommit state and it receives a commit message from the last server in the transaction chain.
- It is the last server in the transaction chain and it has received an acknowledgement message from its replica.

In the event that a read or write operation fails, the responsible server notifies all servers in the transactional chain to abort the transaction.

Chapter 4

Liveness and Safety

In this chapter, we demonstrate the soundness of CrossStitch's protocol and show that in the absence of failures, CrossStitch offers liveness and safety. We use the results in this chapter to demonstrate safety and liveness in the presence of a single server failure in Chapter 5. Informally, the liveness property ensures that the transaction eventually reaches a final state. Specifically, a CrossStitch transaction always commits or aborts and does not indefinitely remain in an intermediate state. The safety property ensures that transactions are isolated and atomic.

4.1 Liveness Properties

In this section, we provide an exhaustive analysis to demonstrate that CrossStitch offers liveness. In particular, we first present the liveness properties of the CrossStitch servers. Given the liveness of the servers, we then demonstrate the liveness properties of the client. We show that a transaction is either committed or aborted.

4.1.1 Liveness Properties of the Server

As Owicki and Lamport define in [32], *"a liveness property asserts that program execution eventually reaches some desirable state."* We first demonstrate the liveness properties of the server. In CrossStitch, final states for a server in the transaction chain include the commit state (CS) and the abort state (AS). We prove that an end server cannot remain in the start state (SS), primary to secondary message sent state (PSMS) or the wait for messages

state (WFM) and that an intermediate server cannot remain in the wait for messages state or the precommit state (PS).

Liveness of the End Server

To show that an end server cannot remain in the start state is trivial since a server does not remain in the start state once it executes its assigned component. Since we are showing liveness properties in the absence of failures, an end server must be able to execute its assigned component. By doing so, an end server may enter the abort state, which is a final state, or it may enter the primary to secondary message sent state. This transition is shown in Figure 4.1

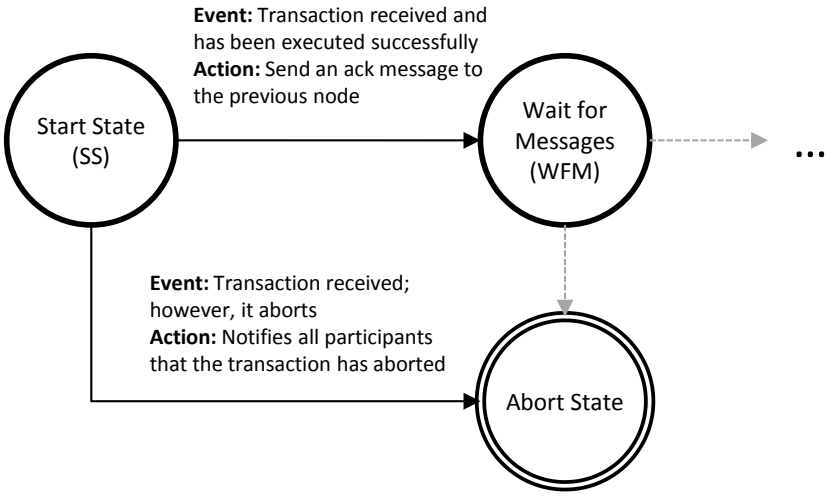

Figure 4.1: Server State Transition from the SS to the WFM

We now demonstrate that an end server cannot remain in primary to secondary message sent state or the wait for messages state. Suppose that the end server, $Server_n$, remains in the primary to secondary message sent state indefinitely. If $Server_n$ is in the primary to secondary message sent state, then $Server_{n-1}$ must be in the precommit state since $Server_n$ requires a precommit message from $Server_{n-1}$ in order to enter the primary to secondary message sent state. Furthermore, $Server_{n-1}$ must be in the precommit state before sending the precommit message to $Server_n$. For a server to enter the precommit state, it must receive a precommit message from its predecessor in the transaction chain. This transition is shown later Figure 4.4. As a result, all previous servers in the transaction

chain were able to successfully execute their assigned component and they must be in precommit state. For $Server_n$ to remain indefinitely in the primary to secondary message sent state, the transaction's replica server must never reply with a primary to secondary acknowledgement message since this would cause $Server_n$ to enter the commit state and send commit messages to all servers in the transaction chain. This transition is shown in Figure 4.2. However, since our model's analysis does not assume server or network failures, the transaction's replica server must reply to the primary end server with a primary to secondary acknowledgement message. As a result, $Server_n$ enters the commit state, which is a final state, and sends commit messages to all servers in the transaction chain.

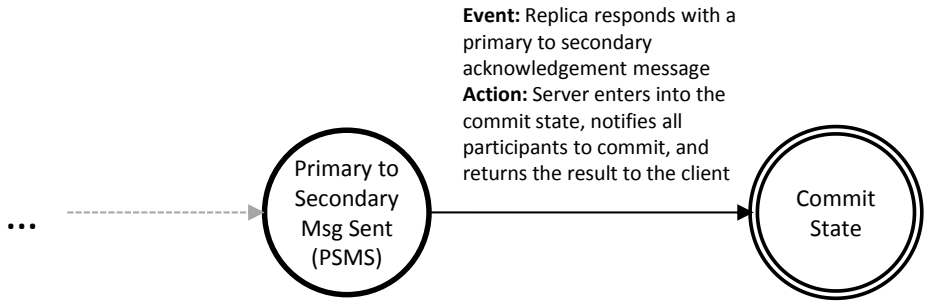

Figure 4.2: Server State Transition from PSMS to CS

We now suppose that the end server remains indefinitely in the wait for messages state. For the end server to remain in this state, it must never receive a precommit message from the previous server. The transition from the wait for messages state to the primary to secondary message sent state is shown in Figure 4.3. Suppose that a precommit message is never sent to the end server ($Server_n$). A precommit message is not sent if the transaction aborts at a server in the transaction chain. However, in CrossStitch's specification, if the transaction aborts at an intermediate server, the transaction is not forwarded to the next server. Consequently, in the case that an intermediate server aborts, the transaction would not have been sent to $Server_n$.

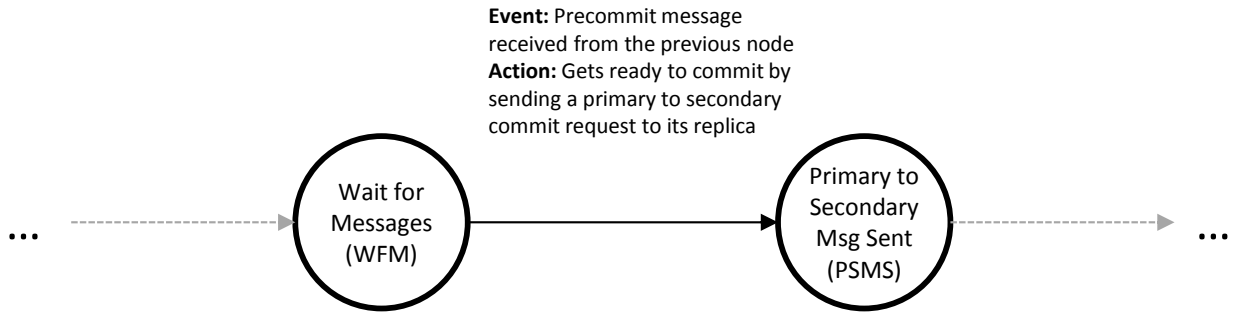

Figure 4.3: Server State Transition from WFM to PSMS

Now suppose that $Server_{n-1}$ never enters the precommit state, hence the precommit message is never sent to $Server_n$. For an intermediate server to never enter the precommit state, it must not have received an acknowledgement message from the subsequent server or a precommit message from its previous server in the transaction chain. The state transitions of an intermediate server are shown in Figure 4.4. Previously, we have considered the case where an intermediate server aborts the transaction. We now consider the case where all intermediate servers successfully complete executing their assigned components. In this case, the transaction is forwarded to subsequent servers in the transaction chain. Since we are demonstrating liveness in the absence of failures, an intermediate server must receive an acknowledgement message from its subsequent server. Similarly, the client receives an acknowledgement message from the first server in the transaction chain. The CrossStitch specification states that the client sends a precommit message to the first server when it receives an acknowledgement message. Therefore, the first server enters the precommit state and sends a precommit message to the next server in the transaction chain. This argument is applied to all subsequent servers in the chain. Eventually, $Server_{n-1}$ receives a precommit message from $Server_{n-2}$ and an acknowledgement message from $Server_n$. By the CrossStitch specification, $Server_{n-1}$ must enter the precommit state.

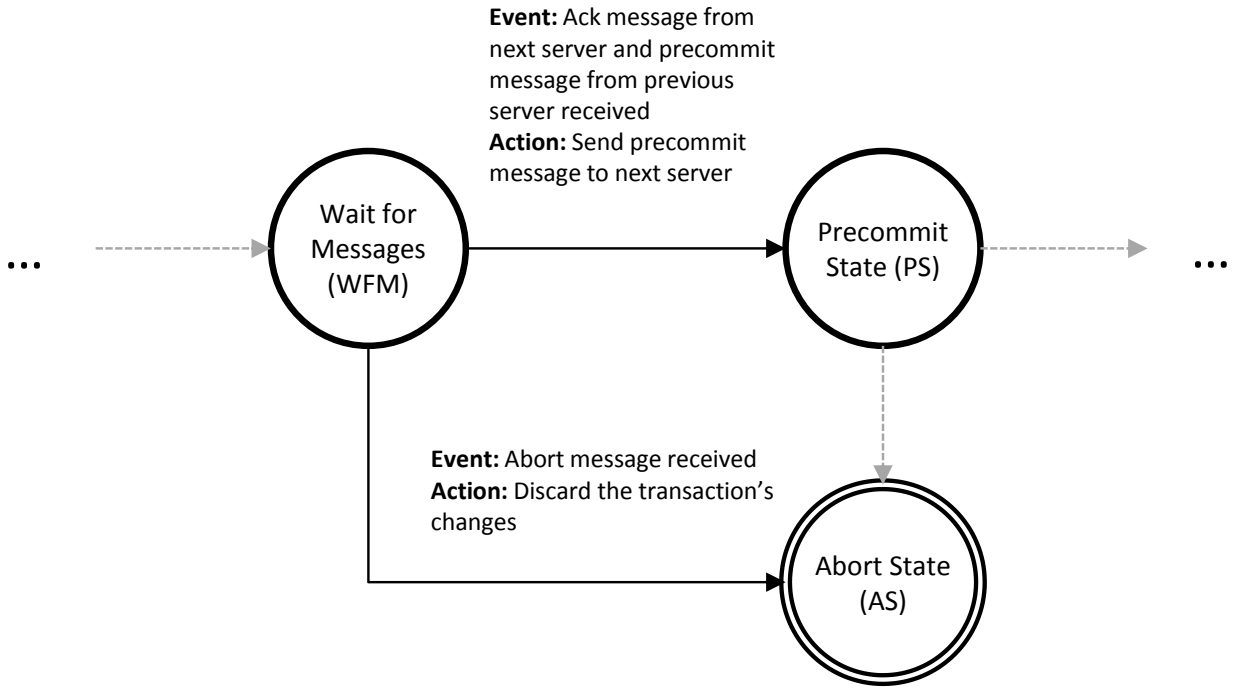

Figure 4.4: Server State Transition from WFM to PS

Finally, we now suppose that an intermediate server does not receive a precommit message from its previous server (or the client if the intermediate server is the first server in the transaction chain). If $Server_i$, where $i \leq (n-1)$, does not receive a precommit message, then $Server_{i-1}$ must not have received a precommit message. Consequently, $Server_n$ remains indefinitely in the wait for messages state and all intermediate servers in the transaction chain must also be in the wait for messages state. Therefore, we now consider the client and the first server in the transaction chain. We have shown that the client and all intermediate servers in the transaction chain must receive an acknowledgement message in the absence of failures. Upon receiving an acknowledgement message, the client will send a precommit message to the first server in the transaction chain. Thus, the first server in the transaction chain will enter the precommit state and send a precommit message to the second server in the transaction chain. Eventually, the precommit messages are sent from server to server in the transaction chain. $Server_{n-1}$ receives the precommit message and enters the precommit state; thus, it will send a precommit message to $Server_n$, which causes $Server_n$ to transition from the wait for messages state to the primary to secondary message sent state. Therefore, an end server cannot remain in the wait for messages state.

Liveness Properties of the Intermediate Servers

We now demonstrate the liveness properties of intermediate servers in the absence of failures. Again, final states are the commit state or the abort state. Therefore, in this section, we show that an intermediate server cannot remain in the start state, the wait for messages state or the precommit state.

An intermediate server cannot remain in the start state since it must execute its assigned component. Upon doing so, it enters the wait for messages state. This transition is depicted in Figure 4.5

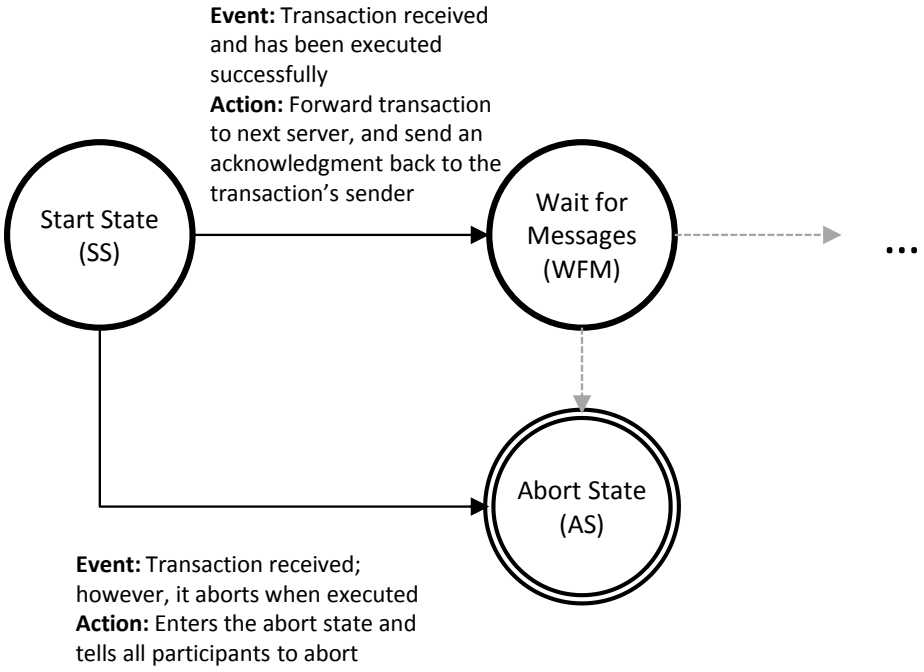

Figure 4.5: Server State Transition from SS to WFM

We now suppose that an intermediate server, $Server_i$, remains indefinitely at the wait for messages state. $Server_i$ is waiting for an acknowledgement message from $Server_{i+1}$ (if it has received a precommit message from $Server_{i-1}$), or a precommit message from $Server_{i-1}$ (if it has received an acknowledgement message from $Server_{i+1}$), or both precommit and acknowledgement messages. Suppose that $Server_i$ is waiting for an acknowledgement message from $Server_{i+1}$. If $Server_{i+1}$ aborts the transaction, then it sends an abort message to all of the previous servers in the transaction chain, which includes $Server_i$. Therefore,

$Server_i$ cannot remain in the wait for messages state if the transaction aborts at a server that is further down in the transaction chain. Now suppose that $Server_{i+1}$ successfully executes its assigned component. Thus, $Server_{i+1}$ sends an acknowledgement message to $Server_i$. In this case, for $Server_i$ to remain indefinitely in the wait for messages state, it must never receive a precommit message from $Server_{i-1}$ (if $Server_i$ is the first hop in the transaction chain, it cannot receive a precommit message from the client). However, as we have shown in Section 4.1.1, the client cannot have received an acknowledgement message from $Server_1$, otherwise it would send a precommit message to $Server_1$, which would cause $Server_1$ to enter the precommit state; thus, precommit messages would be sent from server to server along the transaction chain. However, this contradicts the fact that all servers prior to and including $Server_{i+1}$ were able to successfully complete executing their assigned components. We have now shown that an intermediate server cannot indefinitely remain in the wait for messages state.

Moreover, we suppose that an intermediate server, $Server_i$, remains indefinitely at the precommit state. Suppose that the transaction aborts further down in the transaction chain. As a result, an abort message is sent to $Server_i$. Therefore, $Server_i$ enters the abort state, which is a final state. Suppose that the transaction does not abort. Since transaction chains are finite in length, the transaction will reach its end server. As we have demonstrated the liveness properties for the end server in the absence of failures, an end server will either commit or abort. As a result, either a commit or an abort message is sent to $Server_i$. Consequently, $Server_i$ cannot remain in the precommit state indefinitely.

We have now shown that all servers in the transaction chain enter either the commit or the abort state. Thus, we have demonstrated the liveness property of CrossStitch servers in the absence of failures.

4.1.2 Liveness Properties of the Client

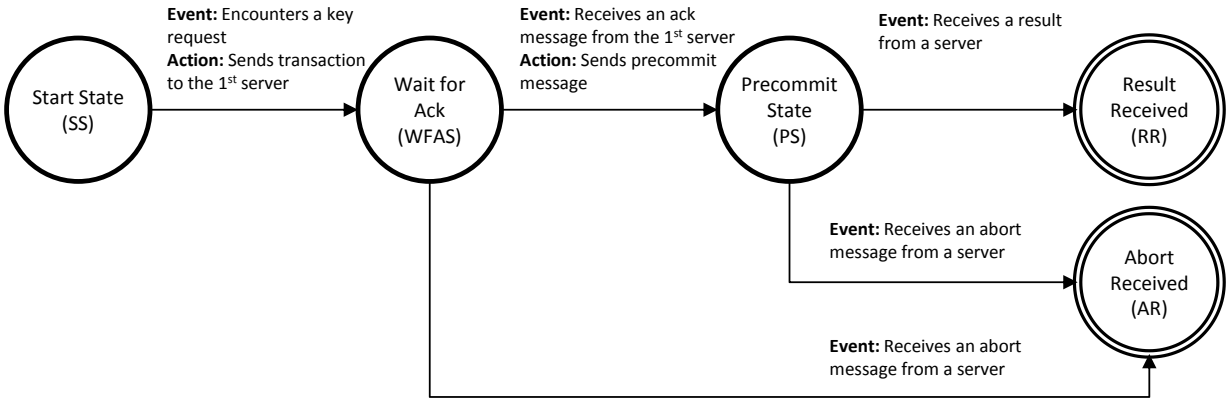

Figure 4.6: Client State Machine

In this section, we now demonstrate that the CrossStitch client offers liveness. Final states for a client include the result received state (RR), which indicates that the transaction is committed, the abort received state (AR). To demonstrate the liveness of the client, we assume that the transaction terminates (i.e. there are no infinite loops). Moreover, we assume the liveness of the servers, which was demonstrated in Section 4.1.1. With these assumptions, we can show the liveness properties of the clients.

As shown in Figure 4.6, the client commences at the start state (SS) whereupon it begins executing the transaction. Eventually, the client encounters a key access and transitions from the start state to the wait for acknowledgement state (WFAS). Since a transaction comprises at least one key access, the client will always make this transition. To perform this key access, the client forwards the transaction to the first server in the transaction chain. The first server in the transaction chain, which is responsible for executing the transaction's first component, replies to the client with an acknowledgement message or an abort message. If the transaction is not successful, it sends an abort message to the client. The client's state transitions from wait for acknowledgement state to the abort received state, which is one of the termination states. However, if the server successfully completes executing the first state, it sends an acknowledgement message to the client. The client state transitions to the precommit state (PS) from wait for acknowledgement state and sends a precommit message to the first server.

Upon entering the precommit state, the client waits for the final result of the transaction to be returned from a server. This result may be the result of the transaction or an abort

message. Since we have shown the liveness of CrossStitch's servers, we can safely assume that the servers will always reply back to the client with the result of the transaction (commit) or with an abort message. As a result, the client will transition into a final state. Therefore, the client will always transition from the precommit state to the result received state or the abort received state and result received state. Since the abort received and result received states are termination states, we have shown the liveness of the client.

4.2 Safety Properties

In addition to the aforementioned liveness properties, we show that CrossStitch also provides safety. In this section, we present properties that are derived from CrossStitch's specification. Using these properties, we show that CrossStitch provides safety. We show that CrossStitch provides transactional isolation and atomicity (i.e., either all servers in the transaction chain commit or all servers in the transaction chain abort). Similar to Section 4.1, our presented safety properties assume that there are no failures. In other words, servers do not fail and receive all sent messages.

4.2.1 Providing Isolation

As mentioned in Chapter 3, CrossStitch uses multiversion timestamp ordering to provide optimistic concurrency control. As shown in [16], the use of multiversion concurrency control mechanisms such as multiversion timestamp order results in serializability. Therefore, CrossStitch provides serializability.

4.2.2 Precommit Safety Property

In order to demonstrate that CrossStitch provides transactional atomicity, we demonstrate the following properties:

- Servers in a transaction chain do not have differing final states. Therefore, if one server in the transaction chain is in the commit state, there cannot exist another server in the transaction chain in the abort state. Similarly, if one server in the transaction chain is in the abort state, there cannot exist another server in the transaction chain in the commit state.

- If the client receives the result of the transaction (i.e., it enters the result received (RR) state), then the end server in the transaction chain is committed and no other server in the transaction chain has aborted.
- If a server in the transaction chain receives an abort message, then at least one server in the transaction chain is in the abort state.
- If a server is in the abort state, then the end server is not in the commit state.

In CrossStitch, once a server enters the precommit state, it cannot initiate an abort for the transaction. As shown in Figure 4.7, a server in the precommit state may enter the abort state if and only if it receives an abort message from a server that is further down in the transaction chain. This property is a result of the CrossStitch specification and is used to demonstrate our aforementioned safety properties. We now show that CrossStitch provides the above safety properties.

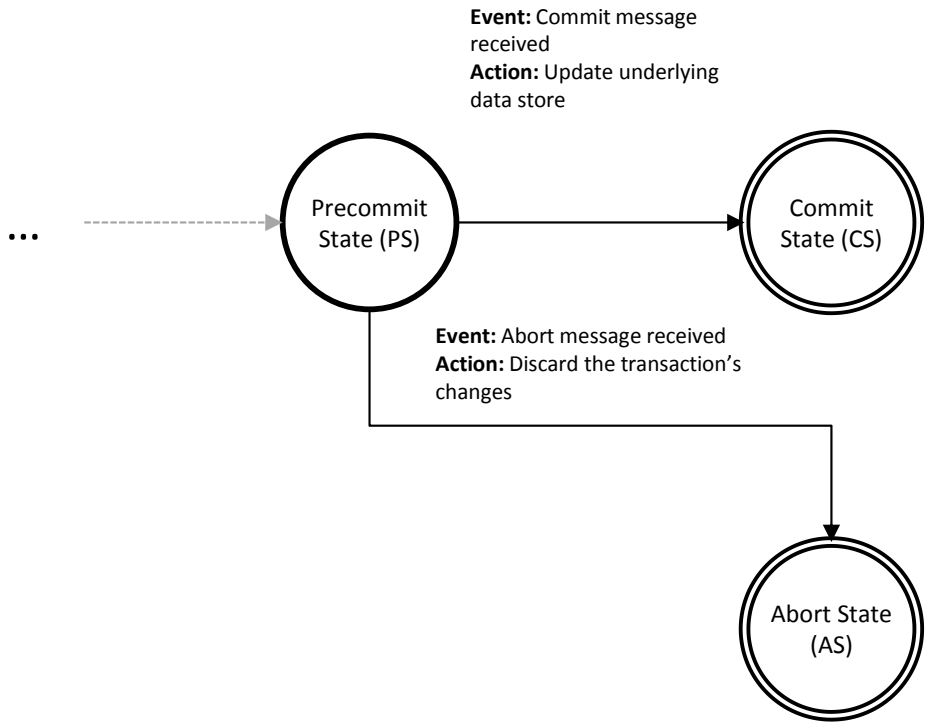

Figure 4.7: Server State Transition from PS to CS

Servers in a transaction chain cannot be in different final states

We first demonstrate that servers, which execute different components, in a transaction chain do not have different final states. We show that if an end server is committed, then every server is in precommit or commit state. Suppose that the end server is in commit state and that there is at least one server, $Server_i$, that is not in precommit or commit state. Therefore, $Server_i$ must be in one of the following states: start state, wait for messages state, or abort state. If $Server_i$ is in the start state, then by definition it has not begun executed its assigned component. As a result, the end server cannot be in the commit state since the transaction's execution is not complete. Furthermore, the end server, $Server_n$, must receive a precommit message from $Server_{n-1}$ in order to enter the commit state. For this to happen, $Server_{n-1}$ must be in the precommit state. As a result, $Server_{n-1}$ also cannot be in the wait for messages state since $Server_n$ would not be able to enter the primary to secondary message sent state, which is the state prior to entering the commit state. Similarly, for $Server_{n-1}$ to enter the precommit state, it must receive a precommit message from $Server_{n-2}$, which indicates that $Server_{n-2}$ is in the precommit state. This argument extends to all previous intermediate servers and the client in the transaction chain. Lastly, suppose there exists at least one server in the transaction chain that is in the abort state. To enter the abort state, a server must receive an abort message from another server or abort the transaction due to a key conflict or user abort, which we call an *abort condition*. In both cases, there exists one server that encountered an abort condition when executing its assigned component. As a result, abort messages are issued to all participating servers in the transaction chain, and the end server never receives the transaction and cannot enter the commit state. Therefore, since all intermediate servers cannot be in the start state, wait for messages state, or the abort state, they must be in either the precommit state or commit state.

End server in commit state and no servers aborted if client receives result

We now show that if the client receives the result of the transaction (i.e., it is in the result received state), then the end server is in commit state and no server in the transaction chain is in abort state. We first suppose that the client received the result of the transaction and that the end server is not in the commit state. By CrossStitch's specification, the final result of the transaction is sent when the end server enters the commit state. Therefore, for the client to receive the result, the end server must have committed the transaction. As shown previously, servers in a transaction chain cannot be in different final states. Therefore, if the client receives the result of the transaction and the end server is in the commit state, no other server in the transaction chain may be in the abort state.

Receipt of an abort message indicates that at least one server is in the abort state

We demonstrate that the receipt of an abort message indicates that one server is in the abort state. Suppose that there are no servers in the abort state and that $Server_i$ receives an abort message. According to our specification, a CrossStitch server generates an abort message if it encounters an abort condition. Upon encountering an abort condition and sending an abort message, it enters the abort state. For an abort message to be received by $Server_i$, there must be a server that is already in the abort state. Therefore, if a server in the transaction chain receives an abort message, then at least one server in the transaction chain is in the abort state.

A server in the abort state indicates that the end server is not in the commit state

If there is one server in the abort state, then the end server is not in the commit state. Suppose that the end server is in the commit state and has committed the transaction. As we have shown in earlier in this section, servers in a transaction chain cannot be in different final states. Since the abort state and the commit state are final states, if there is one server in the abort state, then the end server, which is also part of the transaction chain, cannot be in the commit state.

In this chapter, we have shown that CrossStitch provides liveness and safety. We have shown that the client and every server in the transaction chain eventually enter a final state (i.e., commit state or abort state). Moreover, we have shown that either all servers and the client enter the abort state or the client receives a result and all servers enter the commit state.

Chapter 5

Handling Failures

In this chapter, we describe CrossStitch's failure handling mechanism. We demonstrate that the liveness and safety properties presented in Chapter 4 hold even in the presence of a single failure. Instead of providing durability through replication, we assume that before a server enters the precommit state, it writes its data to durable storage so that it is aware of its state when it recovers[1]. When the failed server recovers, it can query other servers in the transaction chain regarding the status of the transaction and abort or commit the transaction based on this status. We cannot ensure that data on a failed server is available; however, we ensure that the data is available once the failed server recovers. In practice, the use persistent storage to provide availability is not desirable as a synchronous write to disk is expensive; thus, it would not be appropriate for web applications that demand low latency. Although a replication mechanism is more appropriate for low latency applications, we do not describe it in this thesis and leave it for future work.

Our analysis uses a fail-stop model in which servers are aware of other server failures. In this chapter, we describe how CrossStitch uses a timeout mechanism to enable servers to determine their respective course of action in the event of a failure. We also exhaustively enumerate the different failure scenarios and illustrate how CrossStitch recovers from them.

[1]Because our evaluation is based on an in-memory datastore, this overhead is not captured in our result in Chapter 6.

5.1 Timeout Mechanisms

CrossStitch uses timeouts as part of its failure detection and recovery protocol. If a server in the transaction chain does not receive an expected message (acknowledgement, precommit, etc.) after a certain amount of time has passed, it may be the result of a server failure. Therefore, timeouts are used as a mechanism to determine when it is appropriate for a server to query another server that is further down the transaction chain. If a server has not received its expected precommit, acknowledgement, or result messages, then it may begin querying other servers in the transaction chain once the timeout has passed. By querying servers that are further along the transaction chain, a server can determine if a transaction successfully completed and was forwarded. This allows a server to determine whether or not to abort. If the server has entered the precommit state, it cannot abort the transaction as shown in Section 4.2.2. Therefore, the server may now request that another server abort on its behalf. We use this timeout mechanism to provide safety and liveness in the presence of a single server failure.

5.2 Failure Model

In the following sections, we describe how CrossStitch handles server failures. Again, we use the term *intermediate server* to refer to a server that is not the last server in the transaction chain and the term *end server* to refer to the server that is the last server in the transaction chain. Our analysis assumes a fail-stop failure model for our servers. Thus, servers can detect if other servers have failed. For our analysis, we do not explicitly handle network partitions; however, we consider an unreachable server to be a failed server. Therefore, CrossStitch can support a network partition so long as the smaller partition consists of at most one server.

We demonstrate CrossStitch's liveness in the presence of a single failure. Again, final states for a CrossStitch transaction include the commit state and the abort state. We consider client failure, intermediate server failure, and end server failure. For all failure scenarios, suppose that $Server_1$, $Server_2$, and $Server_3$ are three consecutive servers in the transaction chain where $Server_1$ is the first hop. We first consider the case where $Server_1$, $Server_2$, and $Server_3$ are intermediate servers. Our presented analysis can be extended to $Server_{i-1}$, $Server_i$, and $Server_{i+1}$, which are intermediate servers in the transaction chain. Moreover, we also consider the case where $Server_1$ and $Server_2$ are intermediate servers and $Server_3$ is an end server. As CrossStitch uses a transaction replica, we say that

$Server'_3$ is $Server_3$'s replica for the transaction. In this scenario, our presented analysis can extend to any last three servers in the transaction chain.

5.2.1 Client Failure

We enumerate the instances when a client can possibly fail and we describe how CrossStitch can rectify the failure in order to determine the next action. In the figures in this section, an unsent message is equivalent to a message that is not received. We suppose that the client and the servers wish to execute transaction T_i.

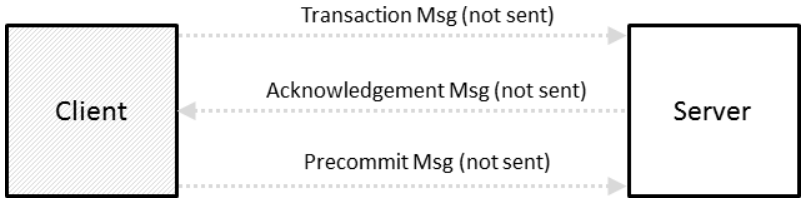

Figure 5.1: Client Failure: Case 1

C1: The client fails before sending the transaction (Figure 5.1). If the client fails prior to sending the transaction, knowledge of T_i does not exist on any server in the transaction chain. Therefore, servers in the transaction chain do not need to abort or commit T_i.

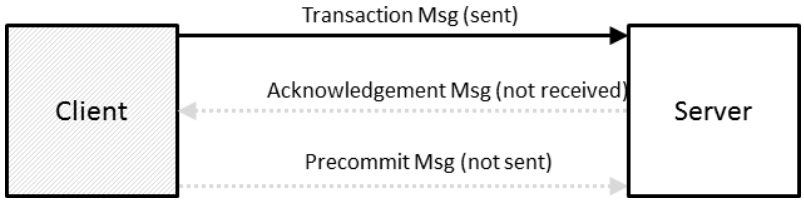

Figure 5.2: Client Failure: Case 2

C2: The client fails after sending the transaction, but before the acknowledgement message is received (Figure 5.2). The client has not sent a precommit message to $Server_1$ since it has not received an acknowledgement message. Without the precommit message from the client, $Server_1$ does not enter the precommit state. Eventually, $Server_1$ timesout while executing transaction T_i and it aborts T_i.

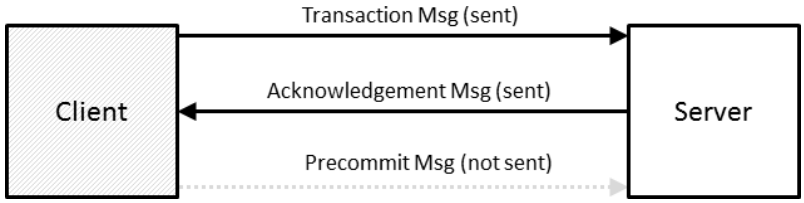

Figure 5.3: Client Failure: Case 3

C3: The client fails after receiving the acknowledgement message but before sending the precommit message (Figure 5.3). $Server_1$ has not received the precommit message from the client; thus, it remains able to abort the transaction. Eventually, $Server_1$ timesout while waiting for a precommit message from the client and it aborts transaction T_i.

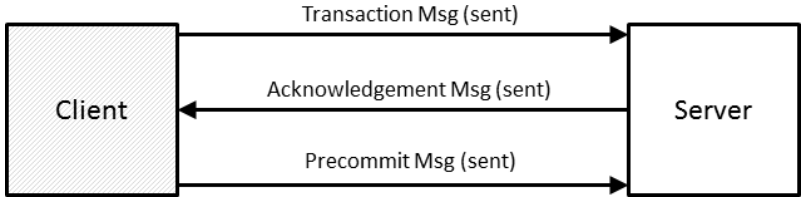

Figure 5.4: Client Failure: Case 4

C4: The client fails after sending the precommit message (Figure 5.4). $Server_1$ receives the precommit message and enters the precommit state. Transaction T_i proceeds normally on the remaining servers in the transaction chain. In the event that the client does not recover from its failure by the time T_i completes, $Server_3$ can maintain the result of the transaction and wait for the client to complete its recovery.

5.2.2 Intermediate Server Failure

We now consider the scenarios where an intermediate server fails. In the following cases, we suppose that $Server_2$ fails. Similar arguments hold if $Server_1$ fails as the messages between the client, $Server_1$, and $Server_2$ are isomorphic to the messages between $Server_1$, $Server_2$, and $Server_3$. We enumerate the instances in which $Server_2$ may fail. We note that we do not assume a second server failure. For our analysis, once $Server_2$ sends a message to another server (e.g., $Server_i$), we assume the recipient ($Server_i$) receives the message.

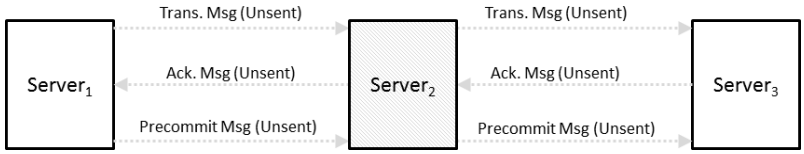

Figure 5.5: Intermediate Server Failure: Case 1

I1. $Server_2$ **fails before receiving the transaction from** $Server_1$ **(Figure 5.5).** $Server_1$ never receives an acknowledgement message from $Server_2$; therefore, after the timeout, $Server_1$ aborts the transaction. $Server_1$ is able to abort the transaction as it has not entered the precommit state.

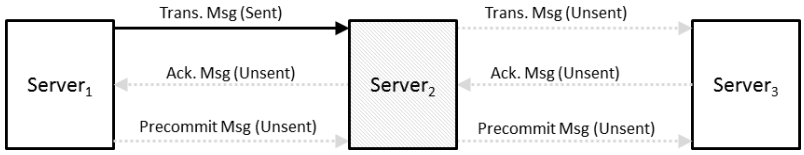

Figure 5.6: Intermediate Server Failure: Case 2

I2. $Server_2$ **fails after receiving the transaction from** $Server_1$**. However,** $Server_2$ **does not send an acknowledgement message to** $Server_1$ **and the transaction to** $Server_3$ **(Figure 5.6).** $Server_1$ never receives an acknowledgement message from $Server_2$; therefore, after the timeout has passed, $Server_1$ aborts the transaction. $Server_1$ is able to abort the transaction as it has not entered the precommit state.

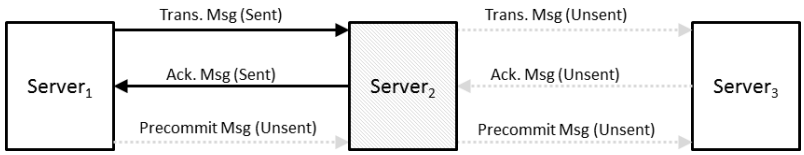

Figure 5.7: Intermediate Server Failure: Case 3

I3. $Server_2$ **fails after receiving the transaction from** $Server_1$ **and sending an acknowledgement message to** $Server_1$**. However,** $Server_2$ **does not send the transaction to** $Server_3$ **(Figure 5.7).** Given that $Server_1$ has received the acknowledgement message from $Server_2$, it is aware that $Server_3$ is the next server in the transaction chain. Since $Server_3$ never receives the transaction, the transaction does not complete. Therefore, $Server_1$ eventually reaches a timeout since it does not receive the result of the

transaction. $Server_1$ queries $Server_3$ whereupon it discovers that $Server_3$ has not received the transaction. $Server_1$ is in the precommit state; thus, it requests $Server_3$ to abort the transaction on its behalf. We note that there is a definite ordering for the sending of transaction and acknowledgement messages; however, the operating system may reorder these messages, so we enumerate all cases in I3, I4, and I5.

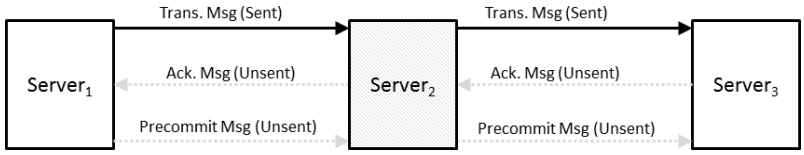

Figure 5.8: Intermediate Server Failure: Case 4

I4. $Server_2$ **fails after receiving the transaction from** $Server_1$ **and after sending the transaction to** $Server_3$**. However, it does not send an acknowledgement message to** $Server_1$ **(Figure 5.8).** Since $Server_1$ does not receive an acknowledgement message, it does not enter the precommit state. As a result, all servers after $Server_1$ do not enter the precommit state. Eventually, $Server_1$ reaches its timeout and queries $Server_2$ only to discover that $Server_2$ has failed. $Server_1$ aborts the transaction by notifying all previous servers in the transaction chain. Similarly, $Server_3$ becomes unable to send an acknowledgement to $Server_2$ and does not receive a precommit message from $Server_2$. When $Server_3$ reaches its timeout, it must send an abort message to all participating servers in the transaction chain. We note that servers that are further along the transaction chain will also timeout. However, these servers will reach the same conclusion regarding the status of the transaction as $Server_3$ and all subsequent servers are not in the precommit state.

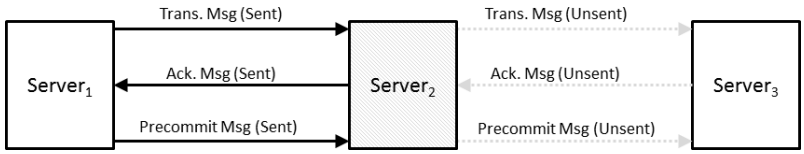

Figure 5.9: Intermediate Server Failure: Case 5

I5. $Server_2$ **fails after receiving a precommit message from** $Server_1$**. However, it fails before sending the transaction to** $Server_3$ **(Figure 5.9).** In this case, $Server_1$'s timeout expires while waiting for a commit or an abort message. $Server_1$ queries

$Server_3$ and discovers that $Server_3$ has not received transaction. Therefore, $Server_1$ requests $Server_3$ to abort the transaction on its behalf.

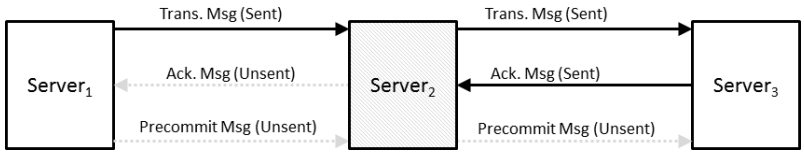

Figure 5.10: Intermediate Server Failure: Case 6

I6. $Server_2$ **fails after sending the transaction to** $Server_3$ **and after receiving an acknowledgement message from** $Server_3$**.** $Server_2$ **has not sent an acknowledgement message to** $Server_1$**.** The timeout for $Server_3$ expires since it never receives a precommit message from $Server_2$. $Server_3$ queries $Server_2$ and discovers that $Server_2$ has failed. As a result, since $Server_3$ is not in the precommit state, it aborts the transaction. $Server_1$ may also abort the transaction as it never receives the acknowledgement message from $Server_2$. Since $Server_1$ is not in the precommit state, it may also abort the transaction.

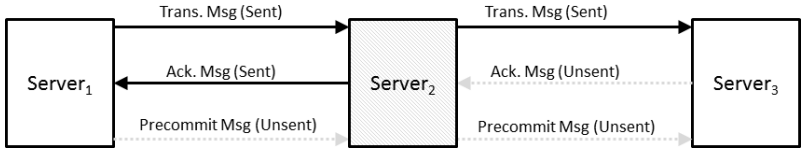

Figure 5.11: Intermediate Server Failure: Case 7

I7. $Server_2$ **fails after sending the transaction to** $Server_3$ **and after sending the acknowledgement message to** $Server_1$**. However, it does not receive a precommit message from** $Server_1$**.** This case is similar to case I5. $Server_1$ eventually queries $Server_3$ after its timeout value has expired. Since $Server_3$ is not in the precommit state, it aborts the transaction on $Server_1$'s behalf.

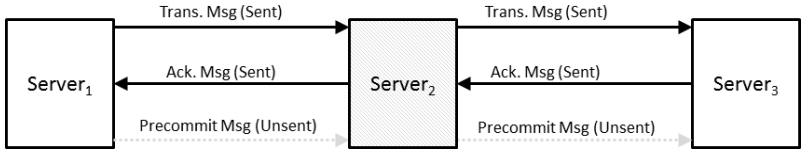

Figure 5.12: Intermediate Server Failure: Case 8

I8. $Server_2$ **fails after sending an acknowledgement message to** $Server_1$ **and after receiving an acknowledgement message from** $Server_3$ **(Figure 5.12).** Since we assume that there is at most a single server failure, $Server_1$ enters the precommit state after receiving the acknowledgement message from $Server_2$. Therefore, this case is similar to I7. $Server_1$ eventually queries $Server_3$ after its timeout value has expired. Since $Server_3$ is not in the precommit state, it aborts the transaction on $Server_1$'s behalf. In this case, $Server_3$'s timeout may expire when it is waiting for the precommit message. As a result, since $Server_3$ is not in the precommit state, it may also abort the transaction.

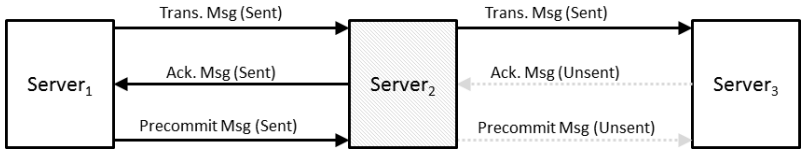

Figure 5.13: Intermediate Server Failure: Case 9

I9. $Server_2$ **fails before receiving an acknowledgement message from** $Server_3$**; however,** $Server_2$ **has received a precommit from** $Server_1$ **and has sent the transaction to** $Server_3$**.** $Server_3$ never receives a precommit message from $Server_2$. Eventually, $Server_3$ reaches its timeout and queries $Server_2$ to discover that $Server_2$ has failed.

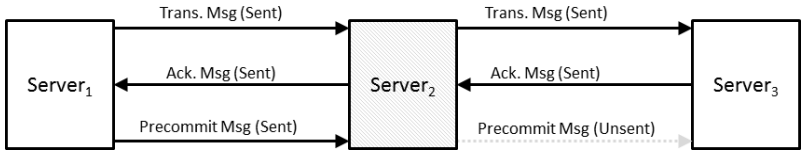

Figure 5.14: Intermediate Server Failure: Case 10

I10. $Server_2$ **fails after receiving an acknowledgement message from** $Server_3$ **but before sending a precommit message to** $Server_3$**; however,** $Server_2$ **has received a precommit from** $Server_1$ **and has sent the transaction to** $Server_3$**.** This case is similar to I7. $Server_1$ eventually queries $Server_3$ after its timeout value has expired. Since $Server_3$ is not in the precommit state, it aborts the transaction on $Server_1$'s behalf. We note that $Server_3$ may also timeout. However, since $Server_3$ is not in the precommit state, $Server_3$ may query $Server_2$ and discover that $Server_2$ has failed. In this case, $Server_3$ may also abort the transaction.

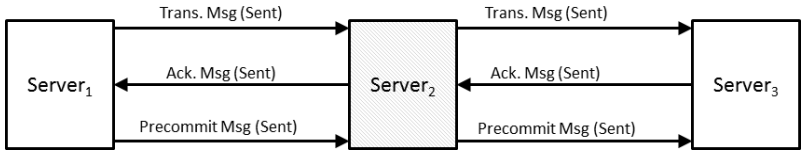

Figure 5.15: Intermediate Server Failure: Case 11

I11. $Server_2$ **fails after sending the precommit message to** $Server_3$**.** Since our analysis only assumes a single server failure, we know that $Server_3$ will eventually have all the prerequisites to enter the precommit state. As the transaction proceeds along the chain, the end server commits the transaction. Since we are assuming a fail stop model, once the end server is aware that $Server_2$ has recovered, the end server notifies $Server_2$ to commit the transaction.

5.2.3 End Server Failure

In this section, we describe the failure and recovery scenarios in the case where $Server_3$ fails. We note that $Server_3$ is the end server in this transaction chain.

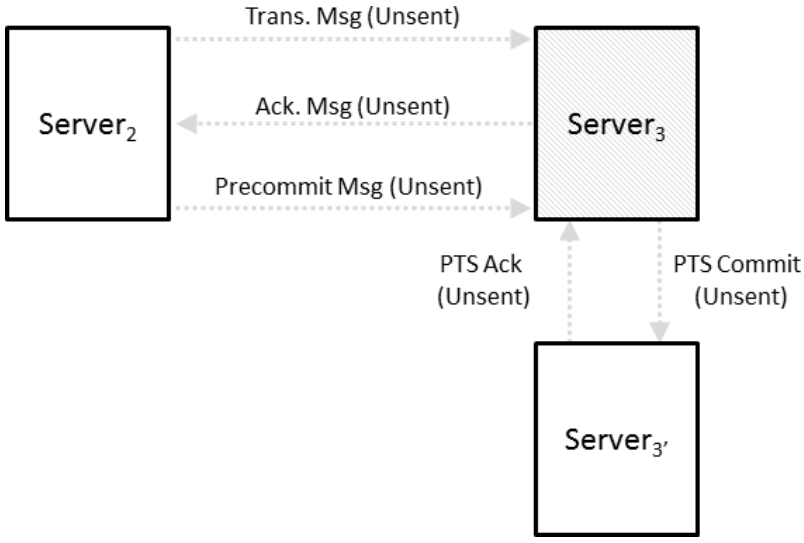

Figure 5.16: End Server Failure: Case 1

E1. $Server_3$ **fails before receiving the transaction from** $Server_2$ **(Figure 5.16).** $Server_2$ becomes unable to send the transaction. Moreover, $Server_2$ does not receive an

acknowledgement message from $Server_3$, which is necessary to enter the precommit state. Therefore $Server_2$ may abort the transaction.

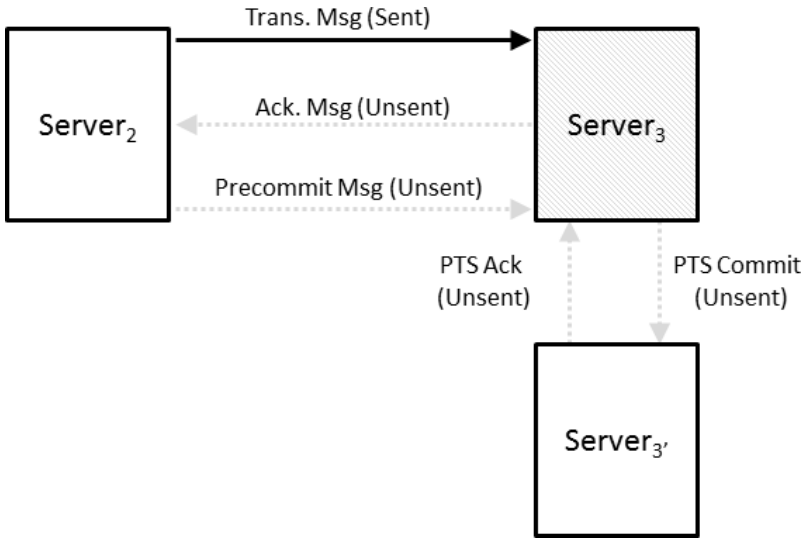

Figure 5.17: End Server Failure: Case 2

E2. $Server_3$ **fails after receiving the transaction from** $Server_2$ **but before sending an acknowledgement message to** $Server_2$ **(Figure 5.17).** Since $Server_2$ does not receive an acknowledgement message, it does not transition to the precommit state. Therefore, after its timeout value has passed, it aborts the transaction.

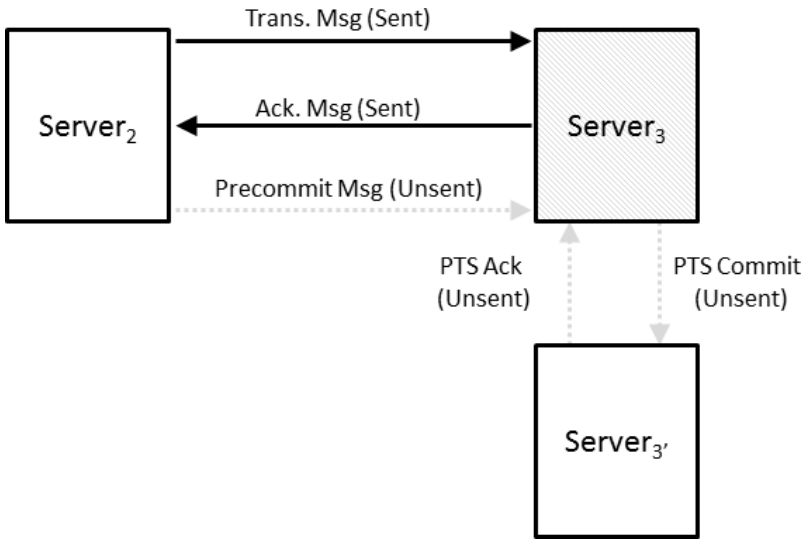

Figure 5.18: End Server Failure: Case 3

E3. $Server_3$ **fails after sending an acknowledgement to** $Server_2$ **(Figure 5.18).** Since our analysis does not include more than one server failure, once $Server_2$ receives the acknowledgement message, it is able to enter the precommit state since it received a precommit message from $Server_1$. As we are using the fail-stop model in our analysis, once $Server_2$ is aware of $Server_3$'s failure, $Server_2$ requests $Server'_3$ abort on its behalf. However, in the fail-silent model, we note that $Server_3$ has the role of the coordinator in two-phase commit. In CrossStitch, we say that a transaction commits once the end server receives a primary to secondary acknowledgement message from the transaction's replica. Since the $Server'_3$ has not sent the primary to secondary acknowledgement message to $Server_3$, $Server_2$ may query $Server'_3$, and $Server'_3$ can abort the transaction on $Server_2$'s behalf.

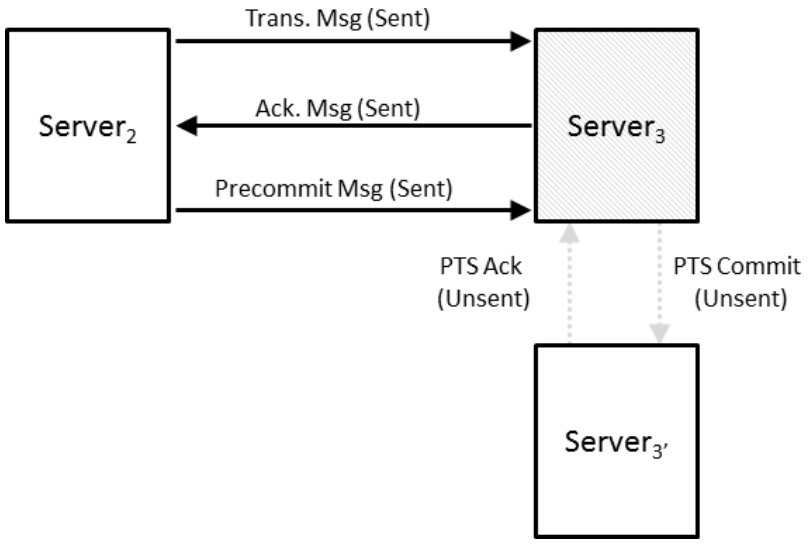

Figure 5.19: End Server Failure: Case 4

E4. $Server_3$ **fails after receiving a precommit message from** $Server_2$ **but before sending the primary to secondary commit message to** $Server'_3$ **(Figure 5.19).** This case is analogous to case E3.

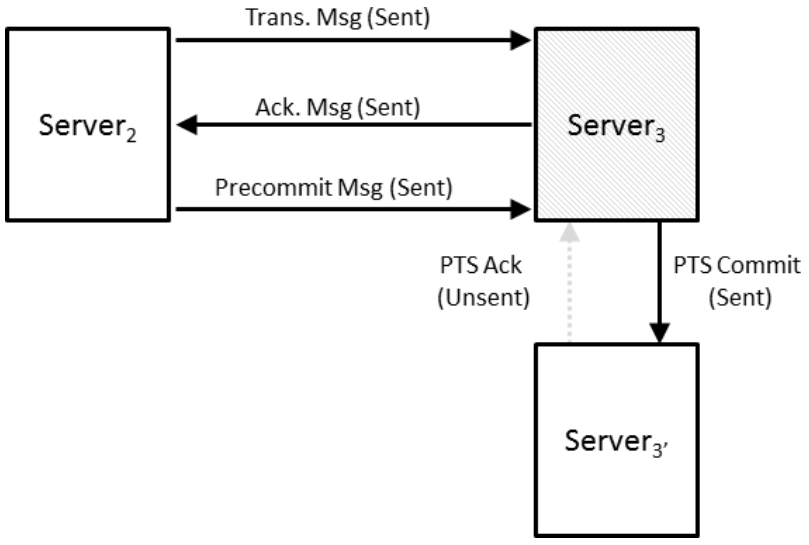

Figure 5.20: End Server Failure: Case 5

E5. $Server_3$ **fails after sending the primary to secondary commit message to** $Server'_3$ **but before receiving the primary to secondary acknowledgement from** $Server'_3$ **(Figure 5.20).** A transaction is committed once the end server, which is responsible for issuing the commit messages and sending the result back to the client, receives the primary to secondary acknowledgement message from its replica. Since it is in the precommit state, $Server_2$ requests that $Server'_3$ aborts the transaction on its behalf. Similar to case E3, in the fail-silent model $Server_3$ has the role of the transaction coordinator. In this case, $Server_2$ and $Server'_3$ must wait for $Server_3$ to recover.

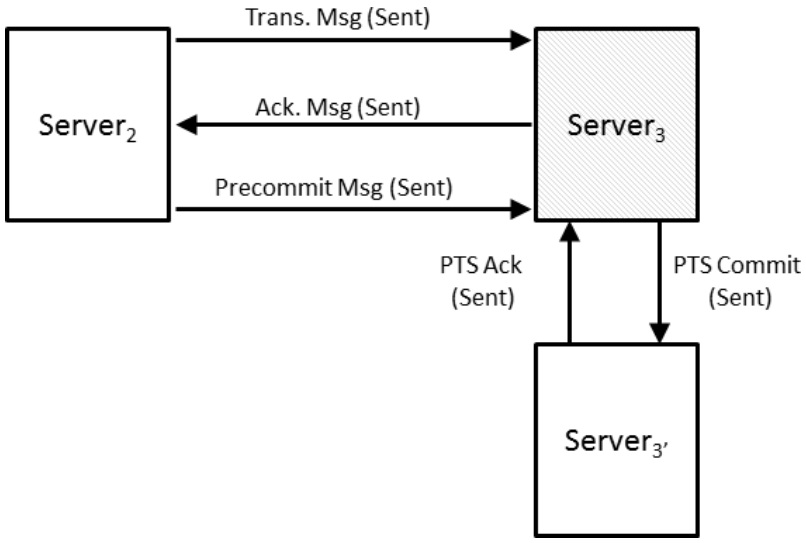

Figure 5.21: End Server Failure: Case 6

E6. $Server_3$ **fails after receiving the primary to secondary acknowledgement from** $Server'_3$ **(Figure 5.21).** At this point, since $Server_3$ has received the primary to secondary acknowledgement, the transaction is considered committed. Therefore $Server_2$ may query $Server'_3$ in order to determine that $Server_3$ has successfully completed its assigned component in the transaction chain. $Server'_3$ notifies $Server_2$ that it has sent the primary to secondary acknowledgement to $Server_3$. Therefore, $Server_2$ may commit the transaction.

5.2.4 End Replica Server Failure

The end server's replica may also fail. In this section, we enumerate the cases where $Server'_3$ may fail.

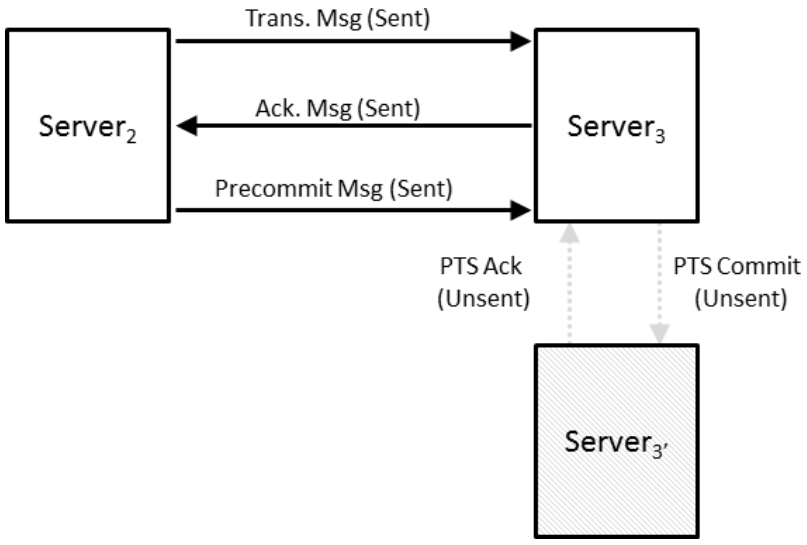

Figure 5.22: End Server Replica Failure: Case 1

ER1. $Server'_3$ **fails before receiving the primary to secondary commit message (Figure 5.22).** $Server_3$ has not entered the precommit state; therefore, it is able to abort the transaction after its timeout has passed.

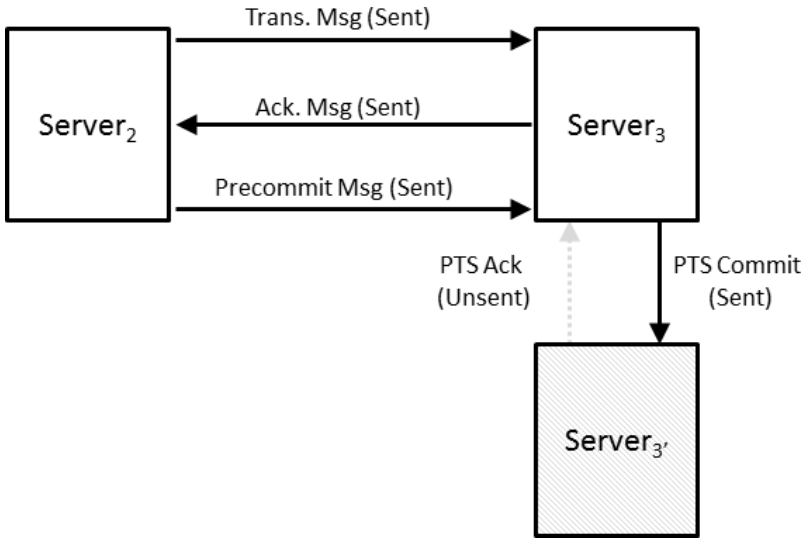

Figure 5.23: End Server Replica Failure: Case 2

ER2. $Server'_3$ **fails after receiving the primary to secondary commit message, but before sending the primary to secondary acknowledgement message (Figure 5.23).** $Server_3$ has not entered the precommit state; therefore, it is able to abort the transaction after its timeout has passed. However, in a fail-silent failure model, $Server'_3$ has the role of the traditional transaction coordinator. Therefore, $Server_3$ would need to wait for $Server'_3$ to recover before it is able to commit the transaction.

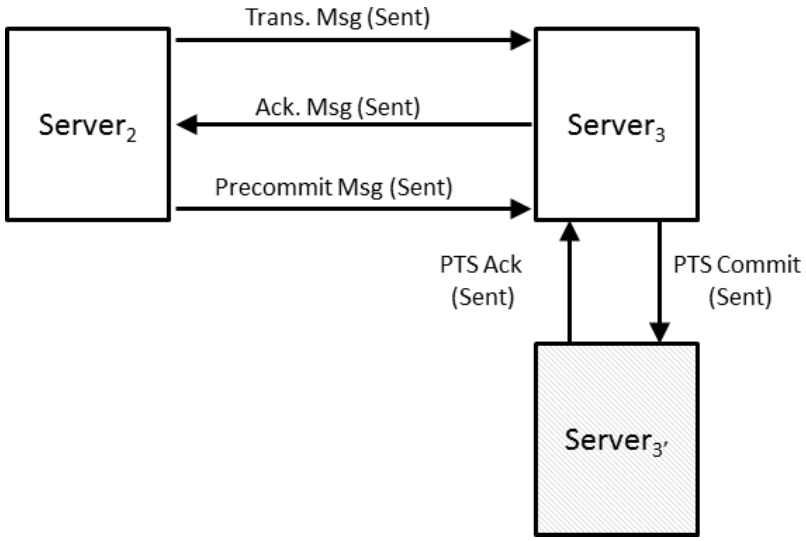

Figure 5.24: End Server Replica Failure: Case 3

ER3. $Server'_3$ **fails after sending the primary to secondary acknowledgement message (Figure 5.24).** Since $Server_3$ has received the primary to secondary acknowledgement message, the transaction is now committed. $Server_3$ can simply notify all participants in the transaction chain to commit.

By enumerating all failure cases, we can show that CrossStitch provides liveness and safety in the presence of a single server failure.

Chapter 6

Evaluation

In this section, we present an evaluation of the CrossStitch transaction processing framework. We vary factors that would affect CrossStitch's performance in order to investigate its performance under different environments and workloads. These factors include the number of concurrent clients, the distribution of key accesses, the length of the transaction, and the read/write ratio of the operations. Firstly, we investigate contention in the presence of multiple clients and significant demand for a small set of keys. Secondly, we analyse the latency of CrossStitch in various scenarios such as different key access distributions and read/write ratios. Lastly, we demonstrate how different key access patterns affect CrossStitch's performance.

6.1 Experimental Setup

In CrossStitch's evaluation, we use eight Intel Xeon machines as servers. Each server has a 3.06 GHz processor, 1024 KB Cache, and 2GB of RAM. Moreover, we utilize a single client machine that has the same specification as the servers. All machines are in the same internal network; thus, our evaluation is representative of an intra-datacenter transaction.

In order to evaluate CrossStitch, we use a dataset that consists of a collection of key-value pairs where each key is 24-bytes. Each key maps to a 1024-byte string. 2,000,000 keys are used in the evaluation of CrossStitch. We use YCSB [14] to generate the keys used in our evaluation.

As CrossStitch uses multi-version timestamp ordering as a means to provide isolation, it maintains multiple versions of data. However, this may lead to excessive memory usage.

In order to reduce memory usage, CrossStitch maintains at most four versions of a single key. Similarly, CrossStitch removes old state machines (i.e., ones that are in either commit state or abort state) that are at least two minutes old. Old versions of data and old state machines are removed in order to limit CrossStitch's memory requirement. Moreover, our implementation of CrossStitch uses an in-memory datastore and does not write to persistent storage. In our implementation, writes are sent to a replica server in addition to the primary server. The replica also sends a notify message to the subsequent server in the transaction chain. As a result, write latency is greater than read latency. On the client-side, CrossStitch is evaluated using a single machine that spawns multiple client applications.

In our evaluation, we show the results using both the CrossStitch implementation that is described in this thesis and an optimized implementation where a server withholds a precommit message in the event of a read-write conflict instead of aborting (as described in Section 3.1.4). Our optimized version also has additional performance optimizations to improve transaction latency including disabling Nagle's algorithm in our TCP sockets. By disabling Nagle's algorithm, messages are sent immediately, thereby reducing latency. Although our optimized implementation includes two major changes, we note that these changes affect different dimensions of CrossStitch's performance and do not have a confounding effect on each other. By turning off Nagle, we reduce the latency of a CrossStitch transaction. Withholding a precommit message in the even of a read-write conflict allows CrossStitch to reduce its abort rate.

6.2 Contention in CrossStitch

We first explore the effects of key contention on CrossStitch. We evaluate CrossStitch using various key distributions including random and various Zipfian distributions. A Zipfian distribution is defined by the probability function as follows: $x^{-\alpha}/\zeta(a)$, where $\zeta(a)$ is the Riemenn Zeta function and α is the distribution parameter.

Intuitively, a greater value of α indicates that the key accesses are more skewed. To provide a representation of Zipfian distribution, we use Python's numpy module to draw 10,000 items. Table 6.1 depicts the number of occurrences that an item of a given rank is drawn, where an item of rank 1 is the most popular item. In a Zipfian distribution, the rank of an item is inversely proportional to the number of times it is drawn. Various studies such as [10, 33] indicate different alpha values for the Zipfian distribution. We select Zipfian alphas of 1.05 and 1.30 to demonstrate the effect of varying key distributions. In

Table 6.1, we find that with an $\alpha = 1.30$, the most popular item occurs a quarter of the time.

Rank	$\alpha = 1.05$	$\alpha = 1.30$
1	494	2563
2	236	1043
3	174	647
4	135	398
5	95	337

Table 6.1: Zipfian Distribution for 10,000 Randomly Drawn Items

Moreover, we explore the success rate of transactions using CrossStitch when factors such as the number of concurrent clients, the distribution of key accesses, and the read/write ratios are varied. The default read-write ratio we use is 80-20 and the default number of concurrent clients is 10. Unless otherwise noted, each experiment consists of 1000 transactions.

In Figure 6.1, we show the number of successful transactions, out of 1000, when we increase the number of concurrent clients. In this experiment, we use a read-write ratio of 80-20. We later show that using an 80-20 read-write ratio results in the greatest abort rate when using CrossStitch; nonetheless, 90% of transactions with a read-write ratio of 80-20 are able to complete successfully when using the optimized version of CrossStitch. Although not depicted in Figure 6.1, we find that a random key distribution, in which there is minimal key contention, results in virtually no failures for up to 20 concurrent clients even in the non-optimized implementation of CrossStitch. Moreover, Figure 6.1 demonstrates that a greater contention for keys, which is reflected in a higher alpha value, results in fewer transactions completing successfully. We find that using the optimized version of CrossStitch, we can significantly reduce the abort rate. For 20 concurrent clients and a data distribution with $\alpha = 1.05$, the number of successful transactions improves from 644 to 938 when using the optimized version of CrossStitch.

In our first experiment, we investigate the abort rate as the amount of key contention increases. We vary the α value for the Zipfian distribution in which a lower α indicates that the data is less skewed. In Figure 6.2, we show that the greater the alpha value, the fewer transactions complete successfully, regardless of the number of concurrent clients. In the above experiment, we fix the chain length of each transaction to be five. Nonetheless, from the results of our optimized implementation, we find that the abort rate is significantly reduced if we wait for a read operation to commit or abort before aborting the conflicting write operation.

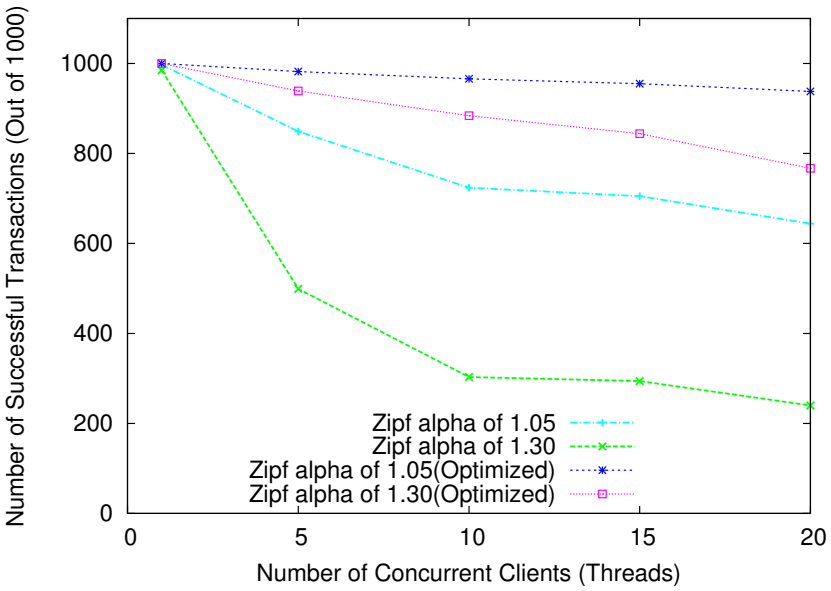

Figure 6.1: Number of Successful Transactions versus the Number of Concurrent Threads

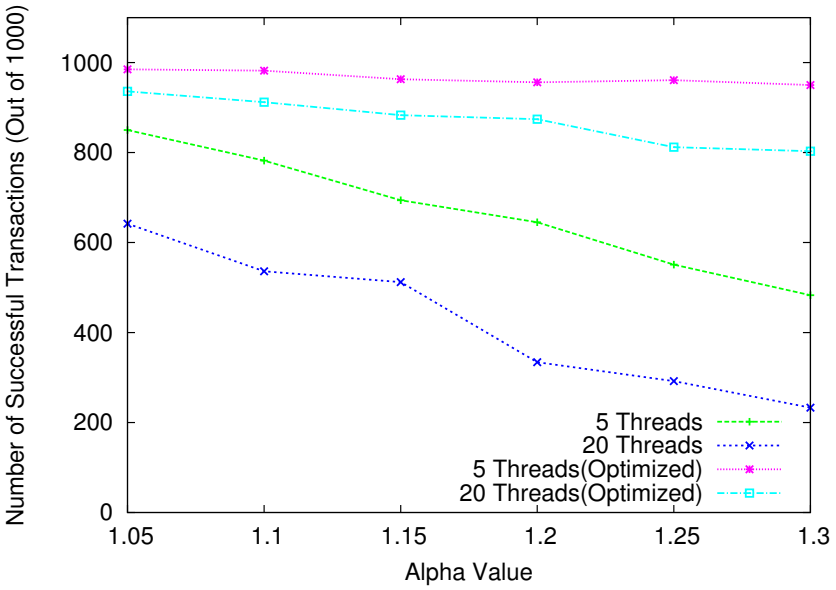

Figure 6.2: Number of Successful Transactions versus Alpha Value

In Figure 6.3, we illustrate the number of successful transactions versus the percentage of read operations. We find that transactions do not abort in an all-read or an all-write scenario since conflicts only occur when a transaction attempts to overwrite a version that has already been read, or when a transaction attempts to read a write that is still pending. As expected, the key access distribution also greatly affects the success rate of the transactions, with a random key access leading to the greatest success rate, since there is little, if any, contention for key accesses. Therefore, we find that the key access distribution and the read/write ratio affect the successful completion rate in CrossStitch.

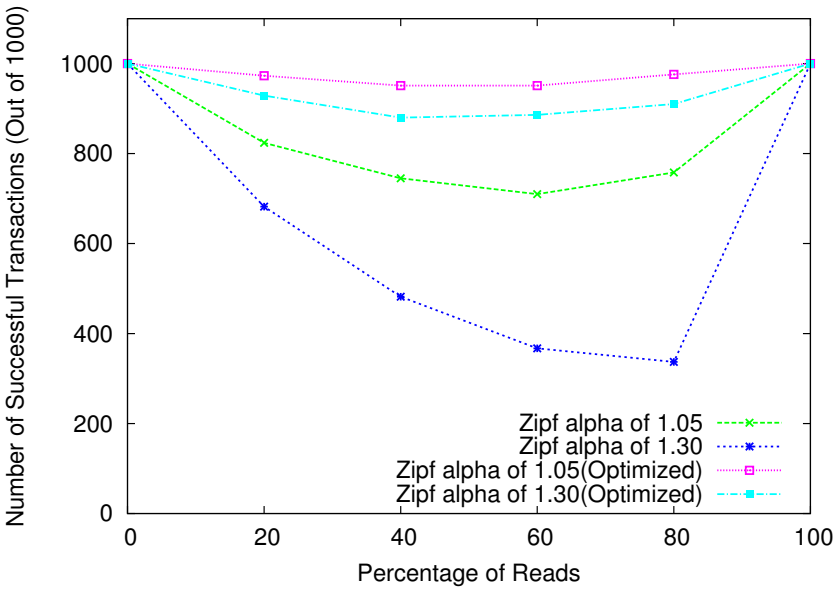

Figure 6.3: Number of Successful Transactions versus the Percentage of Reads

6.3 Transactional Latency in CrossStitch

We also investigate the latency of a transaction in the CrossStitch transaction processing framework, as latency significantly impacts the performance of a web application. We find that the number of concurrent users (clients), the distribution of key accesses, and the read/write ratio also affect latency in CrossStitch. In Figure 6.4, we run 1000 transactions, each with a chain length of five. The transaction completion time is determined by taking the average of the 1000 transactions. We find that transaction completion time increases linearly with the number of concurrent users. However, we find that the rate of increase

is higher when the key accesses are more skewed (in other words, the key accesses follow a Zipfian distribution with a higher alpha value).

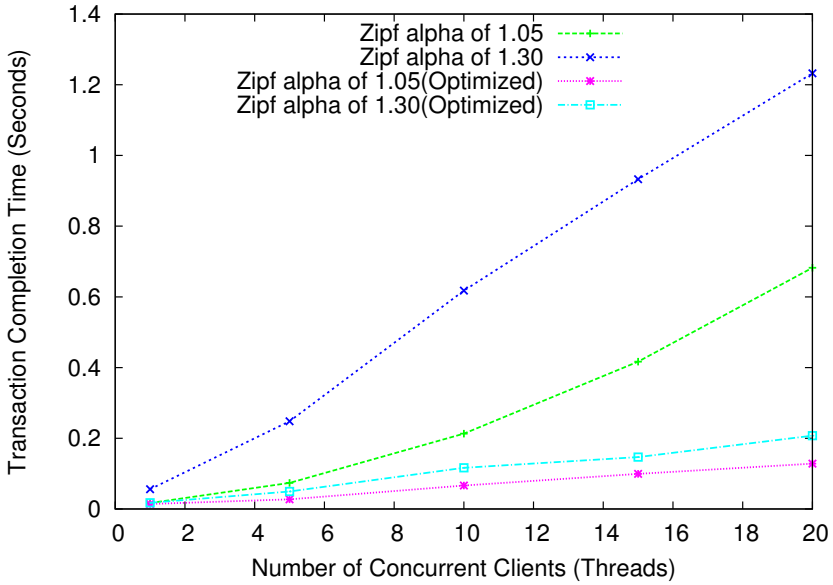

Figure 6.4: Transaction Completion Time versus Number of Threads

In Figure 6.5, we find that if key accesses follow a Zipfian distribution, then the alpha value, which refers to the distribution of data, can have an impact on the transaction's completion time. Regardless of the number of concurrent users, the latency of a CrossStitch transaction increases as the Zipfian distribution's alpha value increases from 1.05 to 1.30. In addition to key access distribution, we find that the read/write ratio of transactional operations in CrossStitch also affect a transaction's completion time, which shows that there is some overhead incurred when executing a write operation.

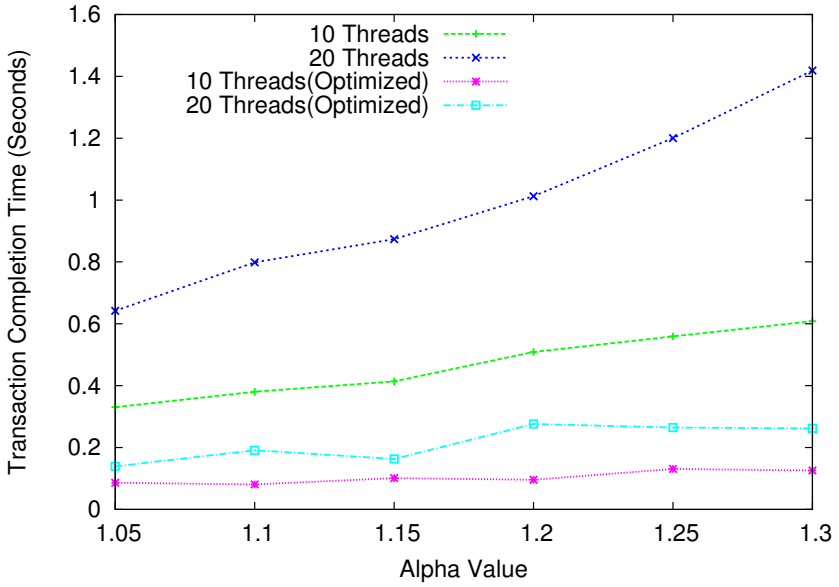

Figure 6.5: Transaction Completion Time versus Alpha Value

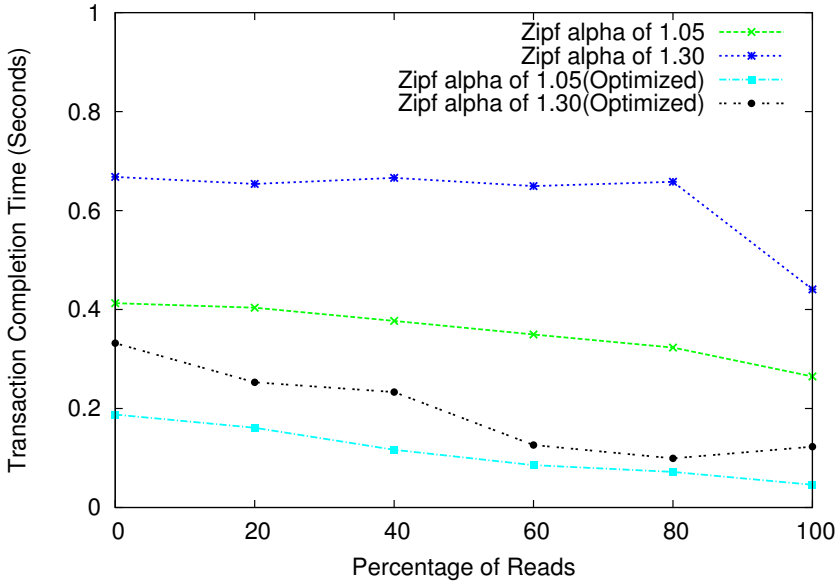

Figure 6.6: Transaction Completion Time versus Percentage of Reads

Figure 6.6 depicts the relationship between transaction completion time (latency) and read/write ratio. As expected, we find that a greater percentage of reads leads to lower latency.

Additionally, we find that the length of the transaction chain also affects the transaction's completion time. This is intuitive as each additional key access is likely an extra hop; thus, the transaction will require more time to complete. In Figure 6.7, we find that the transaction completion time is proportional to the length of its chain.

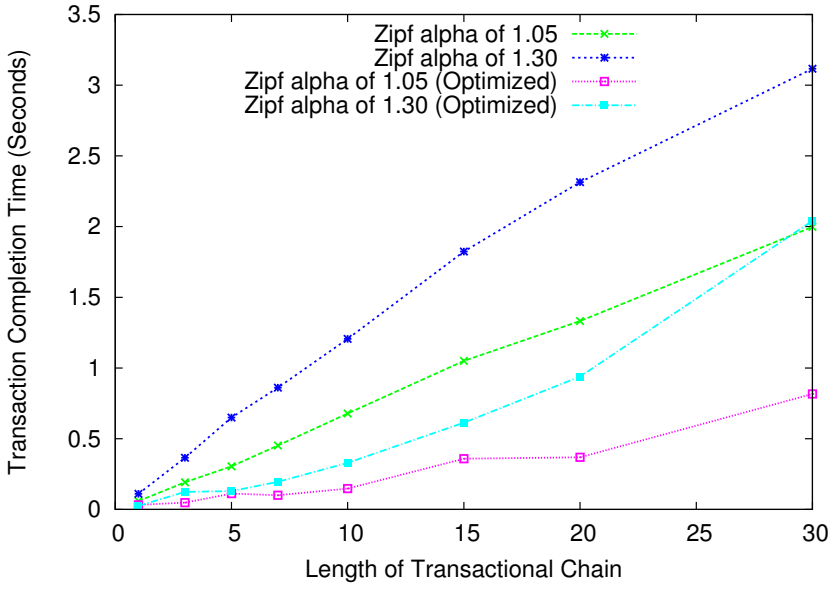

Figure 6.7: Transaction Completion Time versus Chain Length

6.4 Scalability of CrossStitch

In this section, we investigate the scalability of CrossStitch by varying the number of CrossStitch servers. We performed our experiments using Amazon EC2 medium [1] instances as our servers and an Amazon EC2 large instance as our client. We run 5000 transactions to determine the throughput. Each transaction consists of five key accesses, and we use a read-write ratio of 80-20. For these experiments, we use a random key distribution in our experiment in order to better showcase the benefits of using CrossStitch. We omitted the random key distribution from our previous figures in order to reduce clutter.

Moreover, we use the non-optimized CrossStitch implementation. The results are shown in Figure 6.8. We also depict the throughput for a various number of clients. Between 0 to 24 servers, we find that doubling the number of servers results in a 67% to 73% improvement in throughput for ten concurrent clients.

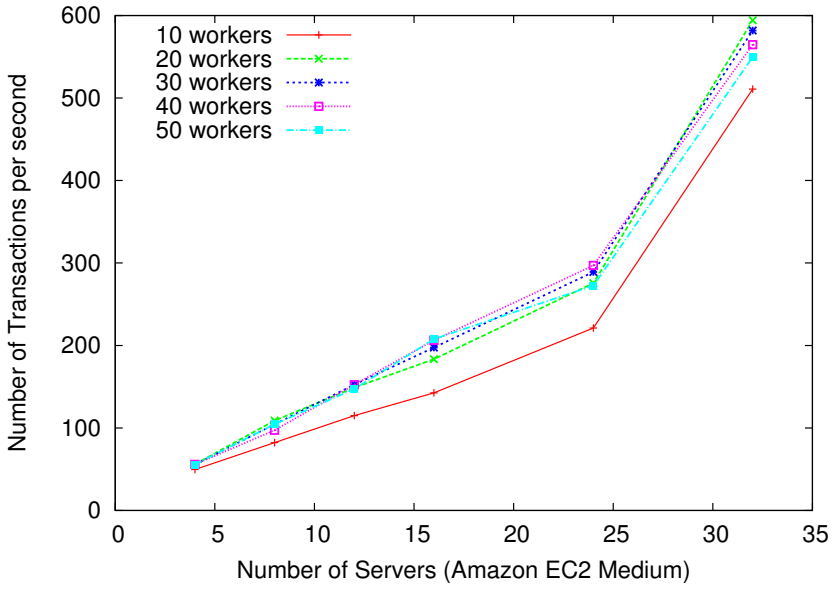

Figure 6.8: Throughput versus Number of Servers

6.5 CrossStitch for Geographically Distant Servers

One of the key benefits of CrossStitch is its effectiveness at serving transactions that span geographically distant servers. In this experiment, we use a read-write ratio of 80-20, 50 concurrent clients, and a transaction chain length of five. Although we use a random key distribution, we configure our transactions to access keys that are located on specific datacenters. Suppose a transaction sequentially accesses key_1, key_2, key_3, key_4, and key_5. In Figure 6.9, the line labelled as *Random* indicates that the transaction's key accesses alternate between datacenters. For example, key_1, key_3 and key_5 are located on one datacenter (i.e., Amazon EC2 servers on the East coast); key_2 and key_4 are located on another datacenter (i.e., Amazon EC2 servers on the West coast). The behaviour exhibited by the *Random* experiment is similar to the traditional transaction processing system where the client is geographically distant from half of the servers. For the line that

is labelled as *Fixed*, they keys are partitioned such that key_1, key_2 and key_3 are located on one datacenter; key_4 and key_5 are located on another datacenter. In this experiment, the transaction only performs a single cross-datacenter hop from key_3 to key_4. We show in Figure 6.9 that CrossStitch has the potential to significantly reduce latency if transactions are designed well. We show that for a transaction chain length of 16 that accesses data on both the East and West coasts, CrossStitch provides a 58 % latency reduction when compared to a traditional transaction processing system.

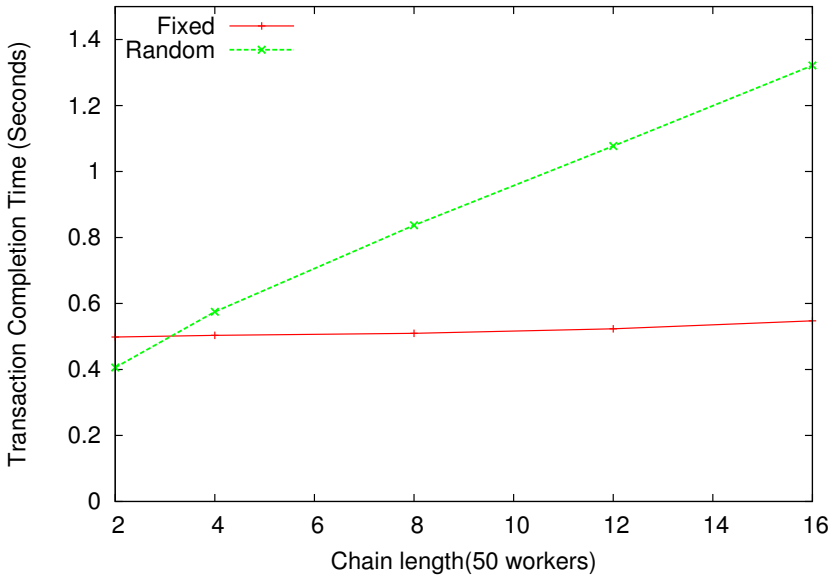

Figure 6.9: Latency versus Chain Length

In summary, we have demonstrated that key contention results in a greater abort rate and higher latency. We have also shown that CrossStitch provides low latency for sequential transactions that are typical of web applications. We find that a transaction that has an 80-20 read-write ratio and contains five key accesses has a latency of $37ms$ to $100ms$ when executing in an environment with ten concurrent clients.

Chapter 7

Conclusion

As the cloud has become the preferred platform for hosting web applications, most commercially available cloud storage systems have focused on scalability and performance. However, transactions have remained critical when performing data operations in order to ensure the integrity of the data that is stored. Therefore, in this work, we presented a new framework, called CrossStitch, for providing transactions for key-value stores, which are the underlying structure for cloud storage systems.

Many commercially available storage systems either do not support transactions or support for transactions is limited within a restricted dataset. Thus, CrossStitch addressed the lack of support for general transactions in today's cloud storage systems. The design of CrossStitch focused on servicing transactions that are typical of web applications. Therefore, we considered the common requirements and characteristics for web transactions. Firstly, in order to meet end-user requirements and expectations, a transactional system must have low-latency. Secondly, web transactions are typically short; their functionality normally comprises retrieving a value for a given key or writing a value in a data store. Lastly, web transactions are highly sequential. The value of a key may be used as a key to access another value.

As a result, CrossStitch was developed to provide web applications, hosted in the cloud, with lightweight and efficient transactions that are able to complete quickly. The novelty of CrossStitch lies in its messaging protocol. Since CrossStitch targets serving web transactions, which are short, CrossStitch sends the transactional code to the servers responsible for executing the key request. Each server that is responsible for hosting a key executes the key request and component of the transaction chain until the subsequent key request is made. Afterwards, the server forwards the transaction, along with the

transactional code, to the subsequent server. Therefore, servers communicate with each other until the transaction is completed. The server that completes the execution of the transaction is then responsible for notifying all servers in the transaction to commit and for sending the reply back to the client. CrossStitch's messaging protocol effectively emulates the functionality of a two-phase commit. Servers that are adjacent to each other in the transaction chain send each other acknowledgement and precommit messages; thus, as the transaction progresses, servers incrementally enter the precommit state.

CrossStitch's messaging framework reduces latency at the client. The client sends only one message to the storage servers and receives a single message that contains the result of the transaction from the servers. If the client is located far away from the server, CrossStitch's transactional framework helps reduce latency as it eliminates back and forth messages between the client and the servers.

We also demonstrated that CrossStitch performs well compared to other systems offering transactions. In particular, we find that CrossStitch scales with the number of concurrent clients that are executing transactions and the length of the transaction chain. Therefore, CrossStitch caters to the requirements and characteristics of web applications and it is able to provide low-latency, lightweight transactions for web-applications.

References

[1] Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types/.

[2] Database Administration Concepts and Configuration Reference. http://public.dhe.ibm.com/ps/products/db2/info/vr101/pdf/en_US/DB2AdminConfig-db2dae1011.pdf.

[3] DB2 Version 10.1 for Linux, UNIX, and Windows English Manuals. http://www-01.ibm.com/support/docview.wss?uid=swg27024478.

[4] Distributed transaction processing: The XA specification. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf.

[5] Distributed transactions (database engine). http://msdn.microsoft.com/en-us/library/ms191440%28v=sql.105%29.aspx.

[6] Oracle database administrator's guide. http://docs.oracle.com/cd/B10501_01/server.920/a96521/ds_txns.htm.

[7] What is Omid? https://github.com/yahoo/omid/wiki.

[8] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 159–174, 2007.

[9] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[10] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM '99).*, volume 1, pages 126–134. IEEE, 1999.

[11] Jake Brutlag. Speed matters for Google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[13] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the First ACM Symposium on Cloud Computing*, 2010.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J.J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally-distributed database. In *Proceedings of the Tenth USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[16] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, Edition 5.* Addison-Wesley, 2012.

[17] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 21–21, 2012.

[18] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS), Volume 38, Issue 1.*

[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2, 2009.

[20] Sudipto Das, Divyakant Agrawal, and Amr 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*, pages 163–174, 2010.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

[22] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 78–91. ACM, 1997.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Nineteenth ACM SIGOPS Symposium on Operating Systems Principles*, pages 29–43, 2003.

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

[25] Jim Gray and Andreas Reuter. *Transaction processing*. Kaufmann, 1993.

[26] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 11–11, 2010.

[27] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[28] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating System Review*, 44(2):35–40, April 2010.

[29] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[30] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research*, pages 123–133, 2011.

[31] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling transaction services in the cloud. *Computing Research Repository*, abs/0909.1, 2009.

[32] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.

[33] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy Web site: Findings and implications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 111–123, 2000.

[34] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Ninth USENIX Symposium on Operating Systems Design and Implementation*, pages 1–15, 2010.

[35] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM SIGOPS Symposium on Operating Systems Principles*, pages 385–400, 2011.

[36] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2007.

[37] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. *ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[38] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[39] Wei Zhou, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing, Special Issue on Cloud Computing, Volume 5, Number 4*, 2012.