Spring 2022

Coordination Protocols for Verifiable Consistency in Distributed Storage Systems

Ashwin Ramaswamy
San Jose State University

COORDINATION PROTOCOLS FOR VERIFIABLE CONSISTENCY IN DISTRIBUTED STORAGE SYSTEMS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Ashwin Ramaswamy

May 2022

The Designated Thesis Committee Approves the Thesis Titled

COORDINATION PROTOCOLS FOR VERIFIABLE CONSISTENCY IN DISTRIBUTED STORAGE SYSTEMS

by

Ashwin Ramaswamy

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2022

Younghee Park, Ph.D.	Department of Computer Engineering
Gokay Saldamli, Ph.D.	Department of Computer Engineering
Harry Li, Ph.D.	Department of Computer Engineering

ABSTRACT

COORDINATION PROTOCOLS FOR VERIFIABLE CONSISTENCY

IN DISTRIBUTED STORAGE SYSTEMS

by Ashwin Ramaswamy

Achieving consistency in a highly available distributed storage system is an impossible task when the system faces network partitions and faulty processes. The complexity is exacerbated in the event of a partition, in which the system allows concurrent processes to send transactions to all the other servers. This updates the current state of data globally which makes achieving replicated consistency challenging. To solve the inconsistency problems, several consensus protocols are used, but have strict requirements in order to make progress and are not guaranteed to ever converge to a single value. Additionally, the coordination required to achieve consistency after a partition will be extremely high as each node must compare transaction times and conflicting data with all other servers in the system. To address the inconsistency in distributed systems, this thesis proposes a new coordination protocol that utilizes four ideas in order for clients to verify the consistency of data: (1) a universal timestamp signatory to certify the global order of events, (2) a relative consistency indicator to determine relative consistency during partitions, (3) an operation-based recency-weighted conflict resolution algorithm to simplify coordination for achieving global consistency, and (4) a rejection-oriented distributed transaction commit protocol to eliminate any guarantees required by atomic commit protocols and verify local consistency. This thesis will evaluate and analyze various issues related to coordination and concurrency under network partitions in order to provide a model for verifiable consistency.

ACKNOWLEDGEMENTS

The process of writing a thesis has been a very informative and explorative journey. I would like to first and foremost thank my advisor Dr. Younghee Park for her guidance and patience in helping me find the best ways to learn about the topics that I was interested in. I really appreciate that she provided me opportunities to expand my knowledge in computer engineering and improve my research abilities. I would also like to thank my committee members Dr. Gokay Saldamli and Dr. Harry Li for contributing their time to help refine my ideas and thesis as a master's student interested in this area of research. Finally, I would like to thank my family and friends who have supported my research endeavors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Distributed storage systems have the challenge of ensuring that all servers are updated to have the exact same state of the data immediately after a user's transaction, which can prove extremely difficult to manage. This means that as the number of servers in a distributed storage system increases, the frequency of transactions between the servers and the network traffic necessary to maintain consistency will also increase [1]. This creates many problems for distributed storage systems that are primarily used for state replication because multiple machines may concurrently attempt to modify a key-value pair while every transaction must be processed by every single machine in the network. If a subset of machines becomes disconnected from the network, it is not possible to have fully replicated consistency among all of the nodes [2], because the disconnected servers cannot be contacted to make updates. According to the CAP theorem proposed by Eric Brewer [3], in the event of partitions in the network, only either availability or consistency of data is possible. In this case, it is the responsibility of the database management system on the server to inform its users that the data may be unreliable due to modifications on servers on the other side of the partition.

The challenge of the associated research lies in the following. In the event that there is a discrepancy of a particular value across multiple servers, there is no existing way to ensure the users that their data is replicated consistently while also being available to them in the midst of a partition. Any discrepancies may occur as the result of concurrent writes without a locking mechanism, uncoordinated writes to a subset of servers in the network, or a Byzantine fault within a server. While consensus protocols aim to converge on a single value, they may not be suitable for post-partition conflict resolution because by selecting one value,

they negate any updates that were made by another server [4]. Specifically, in order to verify that a particular data entry is consistent across the distributed system, the system must be able to uniquely sequence transactions globally, track inconsistency within a partition, mend partitions while preserving consistency and availability, and process distributed transactional operations. While total consistency is not possible to achieve while having an available and partition tolerant distributed system, the current notion of replicated consistency can be improved upon by having the system be able to compare data on a locally replicated level.

Therefore, the purpose of this thesis is to develop protocols to achieve verifiable consistency, such that the user is able to confirm that the available state of data is replicated across all servers, regardless of how recently it was updated on any machine, rather than stale data being available to the client. This will require isolating the components of the distributed system and identifying the unsolved issues of distributed order sequencing, availability and consistency during partitions, post-partition conflict resolution, and concurrent distributed transactions. In particular, this thesis will focus on highly available, multi-endpoint, trusted distributed storage systems, a type of peer-to-peer distributed system that only stores key-value pairs and allows clients to make requests to any server in the network.

The paper will start by focusing on 4 major problems and describe how they hinder the ability of distributed storage systems to ensure the status of consistency of data across the servers. Then it will survey academia to provide a reference of the important research previously conducted. Finally, it provides solutions and the implementations necessary to achieve the ideals stated in the thesis, by laying out the architectural framework needed for algorithms to process data efficiently and contribute to the coordination of the system.

2 PROBLEM STATEMENT

Distributed storage systems need to allow servers to process many transactions simultaneously while preserving a replicated state of data across all servers. However, this leads to many inconsistency issues as servers become partitioned or even fault. Additionally, servers should allow their clients to submit conflicting transactions and autonomously resolve them while verifying that any given key-value pair is confirmed across all replicas. With several hundreds of distributed system architectures, each one is specifically designed to serve a specific purpose. Some make optimizations in concurrent processes for the sake of a slower system while others are more focused on instantaneous accuracy which sacrifice scalability. Regardless of the trade-offs, across all applications, there are four major areas of concern that hinder the system's ability to validate the consistency of data across all servers.

2.1 Distributed Order Sequencing

Ordering events is extremely important so that servers know in which sequence to process requests, some of which may be time-dependent. However, the use of localized clocks has been shown to be inaccurate as minor inconsistencies in timing may accumulate and have a huge delaying effect on the network as a whole. Synchronizing these local clocks involves precise engineering on the atomic level to ensure that across all independent devices the electrical pulses for the clocks occur at the exact same time and that no inconsistencies arise. This is impossible to verify without having another clock to compare with, but another problem arises in how to synchronize that reference clock with the true universal time [5]. Therefore, the question remains: how can time be used to verify the ordering of concurrent transactions across nodes in a distributed system? The problem of ordering is exacerbated as

the number of servers increases. Increasing the traffic of the network will likely increase the latency of messages sent between the servers in unpredictable and non-deterministic ways. Therefore, even if servers try to calculate round trip time or send the current time as a message, the result will not be accurate [6]. For example, a message sent from Server A to Server B at time T may arrive after a message sent from Server C to Server B at time T + 5, which is unintended. Logical clocks such as Vector Clocks and Lamport Clocks are a few among the many solutions conceived to solve the issue of partial ordering, but they all lack the ability to verify that an event universally took place at a certain time. Additionally, logical clocks are unable to order concurrent processes, so a global ordering is not possible [7]. A verifiable solution for this issue is desperately needed in modern distributed systems.

2.2 Availability and Consistency During Partitions

Defying the CAP Theorem has been a focus of distributed systems research ever since it was formalized. However, partitions are an unavoidable problem with all distributed systems to the extent that even the most advanced cloud storage companies advertise their compromises for database availability or consistency to cope with unpredictable network outages [8]. If a provider chooses to remain available during a network partition, then it virtually accepts transactions to update data even though the data may not be consistent across all replicated servers. When a network partition occurs dividing servers into subgraphs of sets of servers, in order for the distributed system to stay active the servers communicate with the clients who wish to update data by allowing reads and writes. Simply reading data from servers does not cause any conflicts since that data should have been verified as consistent when the entire network was connected. Writing data to servers during a partition

is dangerous as the data cannot be replicated to the other side of the partition, and thus volatile. The motivating questions remain: is there any way for each subgraph of the partitioned network of servers to be both available to the clients and consistently represent data across all replicas? Furthermore, is there a way to indicate which data is consistent within the nodes of a partition? That could be useful if by chance all of the clients are communicating to the same partition of the network.

2.3 Post-Partition Conflict Resolution

Since replicated consistency throughout the entire network is impossible in the face of a partition, it is likely that many distributed storage applications allow updates to local servers, resulting in conflicting values. After a partition heals, in order to return the entire system into a fully consistent and replicated state, the inconsistencies in the data are only determined after every single node learns about the state of all other nodes [9]. This raises a concern when the same key has been updated to correspond to two different values by two different servers, because upon the consensus of the entire system, one of the values will be accepted, while the other value is rejected. Therefore, the motivating questions remain: how should the system handle inconsistent updates to the same key in different replicas during a partition? Furthermore, is there an autonomous way to perform a conflict resolution gracefully, without involving the user? Determining that there is an inconsistency is monotonic; as a node surveys that data state of its peers, it only needs to find one inconsistency to determine that the system needs to undergo a mass consensus protocol. No further information will change that fact. Any consensus protocol used needs to take into account the fact that at time T, two different servers may have updated a key in different

ways [10]. Also, if one node updates a key-value pair, the client should expect to see that key-value pair upon request. But if the key is deleted at a later time step by another node in a different partition, should the consensus protocol leave that key-value pair in the final state, or delete it since that was the most recent change? The type of operations, recency of transactions, and status of consistency within a partition are all considerations that a conflict resolution scheme must handle.

2.4 Distributed Transaction Coordination

When more than two servers communicate updates with each other, their goal is to expeditiously propagate a key-value pair to all nodes in the system and receive confirmation that the update is consistently adopted by all. However, a problem occurs when multiple servers concurrently propagate an update to all peer servers at the same time, but the update is received by the different servers at different times [11]. For example, consider a case where one client requests a server to update key ‘x’ to value ‘3’, while another client requests a different server to update key ‘x’ to value ‘4’, at exactly the same time. Under existing distributed transaction protocols, half of the servers may commit to modifying ‘x’ to ‘3’, while the other half may commit to modifying ‘x’ to ‘4’, depending on their arrival time. If the policy of the system is to halt all further transactions until all conflicts are resolved, then this will result in a deadlock which would slow down further coordination of other processes [12]. How can these systems handle distributed transactional updates of different servers? All servers in a peer-to-peer distributed system are considered equal, such that none takes on special privileges or responsibilities, a quality that is often broken in current consensus algorithms which force nodes to take the role of leader, follower, accepter, etc. This is

contrary to the actions of consensus algorithms such as Raft and Paxos or commit protocols like 2-phase commit. Ultimately, the distributed system also needs to consider the fact that these transactions may happen in the middle of a partition and must optimize the data transfer protocol in a way that makes it easiest to determine a value.

2.5 Implications

The previously mentioned problems arise during distributed transactions during and after partitions, and contribute to the issues with consistency, concurrency, and coordination. By addressing these problems, this paper aims to lay transactional guarantees to develop some protocol for each server to verify whether or not all of its data is consistent. The existing solutions of distributed storage systems make use of algorithms and protocols that do not work in suboptimal cases. The several consistency algorithms such as Paxos, RAFT, and Byzantine Fault Tolerance are neither scalable nor reliable in the presence of faults. Also, there is no guarantee of convergence for these algorithms, as proved by Fischer, Lynch, and Patterson. These algorithms rely on electing a leader, but this presents a single point of failure, thereby defying their decentralized nature. After identifying issues with a leader-based consensus protocol, the proposed method in this paper aims to minimize the reliance on a trusted entity to achieve consistency of data after a fault or partition. In some cases, an elected node is unavoidable, so those nodes will need to assert better guarantees to work in the presence of faults and partitions if data is lost in the middle of a consensus protocol [13]. In order to provide a solution for verifiable consistency, this paper will explore the best ways for a distributed system to deal with transactions after and during a partition, and the issues that come with coordination. Additionally, this paper will also provide an implementation of

a distributed system and explain how its state-data functionality along with its time-order functionality, allows it to be both consistent and available as much as possible in the face of partitions.

3 RELATED WORKS AND SURVEY OF EXISTING RESEARCH

3.1 User Intervention as a Solution to Inconsistency

Modern trusted distributed storage systems are best suited for user interactivity in the event of a server failure or a conflicting stored value. When concurrent processes of a key-value storage system commit a conflicting value associated with a key, the distributed system defers control to the user to resolve the conflict, as there is no obvious policy to resolve this issue. This is intermediated by displaying the issue on a user interface on the client side and allowing the user to manually view the conflict and decide on the appropriate solution. For example, Google Docs allows users to concurrently edit text on an interactive online document such that each user is able to see updates made by each other in real-time. In the event that several users are disconnected from the internet and continue to make edits, they will no longer be able to view updates made by their peers in real time. As a result, when they are reconnected to the internet, the Google server will maintain those changes so that the users can decide how to select the intended value. This is done so that no party's work during the network outage is reversed. Similarly, the "merge conflict" functionality of Git, allows users to manually parse conflicting lines of code and resolve them, when Git is unable to determine the correct sequence or priority. Delegating control of conflicts to the user eliminates the overhead of consensus required by the distributed system, thereby allowing it to manage other processes and transactions.

3.2 CRDTs as a Utility to Solve Inconsistency

An autonomous solution to this issue without involving human interaction is the Conflict-Free Replicated Data Type (CRDT). CRDTs are data structures replicated across

multiple servers that can be updated independently and concurrently, such that coordination is not necessary between machines. Once each server receives the conflicting transactions, the data structure reconstructs the values in such a way that inconsistencies can always be resolved without compromising any of the data [14]. Their functionality was later enhanced by Martin Kleppman [15] in his research for collaborative text in 2018. Kleppman uses idempotent operations to aggregate different transactions and achieve consistency among different transactions occurring at different times. In this way he is able to achieve unambiguous transaction order. However, he remarks that "generating concurrently editable data objects is an unsolved issue". The proliferation of different types of CRDTs, demonstrates the breadth of options available, demonstrating that this is well-suited to handle post-partition conflict resolution. However, the inability for CRDTs to have all parties converge upon a singular value shows that they are not a suitable solution for key-value storage databases. However, the underlying data structure may be modified or repurposed to suit the needs of any distributed application. There is clearly a limit to how far research in conflict-resolution distributed systems has come, indicating that alternate data structures are necessary.

3.3 Consensus Protocols as a Utility to Solve Inconsistency

Consensus protocols are a category of distributed algorithms that are used to coordinate between the servers of a distributed system and converge on a final result. Consensus is particularly needed in distributed storage systems to ensure that all of the servers have identically replicated data states. Additionally, all consensus algorithms must eventually terminate and select a meaningful proposed value, such that the selected value is

one of the proposed values. These requirements are often called safety and liveness properties of consensus [16]. Election algorithms are the most common form of consensus because they involve electing a node to manage a direct selection of data converging on a single value. This usually involves delegating certain privileges and responsibilities to certain nodes to ensure that they play a part in the coordination of proposing and accepting data values. The most common consensus protocols are Paxos, Raft, and PBFT which have been used for decades. However, these processes may not be the best choice for conflict resolution convergence after faults, because they rely on using a trusted entity by appointing 1 leader to assess each discrepancy and determine a valid result [17, 18]. In the event that the leader faults and starts selecting invalid values, the integrity of all databases could be compromised. Leslie Lamport introduced the Byzantine Generals problem as a thought experiment in the unexpected behavior of servers. As a result, some distributed systems account for Byzantine Fault tolerance as a correction to complications that may arise from these unexpected problems, of which the most common algorithm is PBFT [19].

Alternatively, Nakamoto consensus is a non-deterministic form of consensus that is used in peer-to-peer permissionless blockchain applications in which each server races to propagate their value to as many peers as it can. This form of consensus aims to converge upon a value through probabilistic acceptance by nodes, in which each verified block in the blockchain is appended to the previous if it is the first block propagated. Eventually, all nodes will have the correct order of blocks [20]. This is extremely useful from a security perspective as it is used to deter against Sybil attacks. However, in cryptocurrencies like Bitcoin, the eventual consistency of the verification can take up to an hour to ensure that the

proposed value has been accepted onto the blockchain, and that all blockchains are the same. Similarly to Paxos, there are no guarantees for having all nodes contain the same blockchain, just the probabilistic expectation of convergence [21]. Therefore, this is not suitable for trusted distributed storage systems. However, this autonomous procedure can be used as inspiration to achieve collaboration between all of the nodes without the use of an elected signatory or a designated party to manage discrepancies.

3.4 Sharding as a Utility to Solve Data Availability

Sharding attempts to solve data availability by taking advantage of statistical multiplexing and the idea of spatial locality and splitting the database of one server into multiple databases spread across several servers. It is useful due to the idea that data modified by one client for a particular database is likely related to subsequent data so information retrieval will be easy [22]. Therefore, in the event of a partition or an outage, the client will be able to retain some data by accessing an available server. However, the major disadvantage is that if the entire database is lost then the client cannot use any of the data. Therefore, sharding stores the data among many different servers and is retrieved when needed. This can further be improved through the use of consistent hashing and chain replication [23]. However, availability of data is not the issue due to the proposed system architecture if a RAID-10 memory layout is used, as it forces each server to maintain a copy of the data of all states, namely a universal state. The main pitfall of sharding lies in the fact that by distributing parts of the data across multiple servers, any faults in any server would require a heavy reliance on interconnection between adjacent servers. If sharded data is not

replicated, then by losing 1 server, the data can only be reconstructed partially. However, this idea of replicating data can be extremely useful for users during partitions.

3.5 Academic Analysis of the CAP Theorem

The 1990 paper by Fischer, Lynch, and Paterson showed that it is impossible to achieve distributed consensus in the event of even a single faulty process, for example a partition or a Byzantine failure [1]. However, this paper proposes that the system can continue to make progress by storing the state information and sharing it during the coordination stage. Eric Brewer also introduced the CAP theorem in 1999, which was later formally proved by Seth Gilbert and Nancy Lynch in 2002 [8]. Brewer explained that in a distributed system that is partition tolerant, the system can accommodate only either consistency of data among the servers or availability for updates by the client, but not both. This is the fundamental framework for distributed systems as most are customer facing, so most have engineered solutions to improve the user experience on one of the two options. All consensus algorithms and state replication protocols are constructed with the CAP theorem in mind, acknowledging that in the event of partitions, the state of the system must be preserved in some way. This thesis will demonstrate how the proposed ideas in section 5 can enable a distributed system to achieve high availability and consistency upon network healing. General distributed consensus was tackled as well by Lamport with the introduction of Paxos. Raft was later introduced by John Ousterhout and Diego Ongaro as a Paxos alternative that is reliable and fault tolerant. Both algorithms have effectively the same performance and guarantees to achieve the same result. Lamport also formalized the idea of sequential ordering and logical time in distributed systems [24]. However, he noted that a

synchronization of clock devices is insufficient due to inconsistencies and clock drift over time. Therefore, he predominantly used logical clocks but noted that this will at most provide a partial order for concurrent processes. Hellerstein and Alvaro's CALM theorem describes how monotonicity in distributed transactions can be used to achieve distributed consistency [25]. These contributions to the field of distributed systems greatly expanded the knowledge that researchers now have about the abilities and limitations of distributed computation and the network infrastructure that supports it. By formalizing technological ideas in mathematics and providing proofs, researchers now have a quantifiable way to measure different metrics and work to improve parameters.

3.6 Issues with Distributed Transactions

Distributed transactions must comply with the ACID properties of distributed databases in order to ensure that they do not leave the database in an unconfirmed, inconsistent, and unsalvageable state. First, the transactions must be atomic, such that any update to the database is processed in its entirety or not at all. There can exist no state in which a transaction only partially modifies the state of a database. This is because non-determinism and uncertainty always exists when dealing with networks, so in order to allow databases to rollback to different states and compare different values with other databases, having discrete states is the only way to do this. Secondly, the transactions must be consistent (or correct), such that no transaction will cause the database to be in an invalid state. Third, the transaction must be isolated, such that any transaction executed concurrently would affect the database in the same way if the transactions were executed serially. This is important so that the system can ensure that no concurrent transaction outcome would interfere with the

data of any other concurrent transaction outcome [26]. Finally, any confirmed committed transaction can also be confirmed as persistent, ensuring that in the event of a fault immediately after the commit, the transaction does not disappear from the database, which would cause it to be inconsistent from other databases that have committed the transaction but have not faulted. In order to ensure the satisfaction of all these criteria on both the coordination end and the database end, various commit protocols, like Two-Phase Commit, have been introduced and implemented for distributed Transactions.

Furthermore, scalability for distributed systems suffer the problem of exponential message increase for coordination among the nodes. Therefore, it is necessary to store some data value that actively represents the state of a particular transaction across the entire topology [27]. This provides information to the user in the event of failures and solves so many problems, rather than relying on the simple key-value storage. In this distributed systems model, the dirty bit is improved to incorporate the state of the local network, which will then be compared globally to assign priorities.

The peer-to-peer model requires each node to send every user update to all other nodes. Therefore, what happens if before commitment, one of the nodes faults? Many distributed applications have the policy of continuing transactions with other servers to maintain replicas for backup purposes. Having the state of the local topology stored along with the data provides information to the user in the event of failures and solves so many problems, rather than just key-value storage. The distributed algorithm can use this information to make decisions, rather than just relay the issue back to the user. Consider a hypothetical scenario where there exist 3 e-commerce web servers that service requests from

many clients simultaneously, they all are trying to communicate to 3 replicated databases. For a read-only system, there will never be a consistency issue. But for a read-and-write system, the problem occurs with the coherent memory view [28]. What happens if a read occurs immediately after a write, with no time for updating all 3 databases? Solving concurrency issues like the one described above is the basis for redesigning a distributed system architecture that accommodates post-fault convergence of data.

4 SYSTEM ARCHITECTURE

A distributed storage system specializes in two distinct components. First, it must have a distributed system architecture in which each server distributes data updates across all of the servers. Secondly, it must have functional storage system capabilities, allowing clients to store, update, and read data from any server. In order to implement a solution to the stated problems, it is necessary to design a distributed storage system architecture to detail the communication and storage requirements for how data will be proposed by transactions. In particular, this thesis aims to optimize a specific subset of distributed systems, namely "trusted", "multi-endpoint", "highly available", "key-value" distributed storage systems. This section will present the proposed methods to satisfy the main requirement under other assumptions to build up fault tolerant and trusted distributed systems.

4.1 Requirements of the Desired Distributed System Architecture

First, the architectural requirements of trusted distributed systems will be presented. Primarily, this will require that the communication channel between the servers be secure and not susceptible to Man-in-the-Middle attacks. In a practical distributed system implementation, this can be satisfied by encrypting the traffic, assigning digital certificates, and even pre-delegating servers to participate in the network. For the purposes of this analysis, this level of security can be abstracted, as the security mechanism is not important as long as the servers can communicate. This begets the assumption that no server will be malicious and will accurately follow any protocol without any mistakes unless it faults. The constructed distributed system for this thesis will not include any security mechanisms, functionally behaving identically to a trusted system.

Next, the architectural requirements of multi-endpoint distributed systems will be presented. This model is defined by the interactions with clients and distinguishes itself from single-endpoint distributed systems. Single endpoint distributed systems are mainly used for storage and replication purposes, in which clients communicate with only one server that is designated as active. The other servers are replicated for backup and do not communicate with the clients, only communicating with the active server. The active server propagates all updates, transactions, and queries to all backup servers in order to achieve consistency. This model is useful because in the event that a backup server fails, no data will be lost because the other backup servers can be used. Backup servers provide no functionality other than to store replicated data. If the active server fails, then the backup servers can merely stage an election to activate one of the remaining servers as the new primary host for the clients. Furthermore, the simplicity of having all clients bottlenecked through one single server forces consensus to be achieved because that server exhibits authority over the final state of the data. However, the significant disadvantage of this model is that it creates a single source of failure. Consequently, the multi-endpoint distributed system exists such that all servers in the system are equally available to clients for interfacing with the data. This still requires a universally replicated state of data such that all servers are equal while designated as active.

Thirdly, a highly available distributed system is one in which the servers of the system are always available to the clients for processing transaction requests regardless of the state of partitions within the system. This does not include servers that have experienced faults, and instead includes servers that are active and working. This is any server that has both read and write abilities to its internal database and communicate with the client and

make updates in real-time to its own local database. The server does not necessarily need to be able to communicate with any subset of servers. In the event of being partitioned from the rest of the servers, by remaining available to the clients, any data updates are not guaranteed to be consistent with the rest of the system. A fault may be the result of network connectivity issues, or hardware/software failures within the server itself. Regardless of the issue, this may cause network partitions if the topology of the servers becomes split.

Finally, a key-value distributed storage system is one that accepts two components of data from the user, namely one locally unique ‘key’ entry and one unrestricted ‘value’ entry. These will then be mapped together and stored in the servers’ databases in a way that allows the user to request both data components by referencing the ‘key’ data. The proposed design of the database will in actuality contain several other columns in addition to the “key” and the “value” data in order to save machine state information. This will be heavily detailed in section 4.2.2. The goal of the proposed methods, introduced in section 5, will be to provide a verifiability of consistency utilizing the architectural model of the distributed system depicted in this section.

4.2 Design of the Proposed Distributed Storage System

This section presents the necessary design requirements in order to implement the proposed method, by laying out the client-server communication model, the storage system design, and the operational capabilities of the distributed systems. Based on these design principles, this paper presents four different methods to verify consistency in distributed systems in the following sections.

4.2.1 Client-Server Architecture Implementation

In order to demonstrate the coordination and modification capabilities between the nodes, it is necessary to design a simple database that displays in real time how distributed transactions are being transmitted between servers, the order in which servers are storing them, the way in which the clients indicate to the server of a new transaction, and how fast the updates are implemented in the database. The construction of the distributed system can be abstracted as the communication between individual nodes, called servers. Most programming languages support the client-server architecture in which sockets are bound to an IP address and listen for incoming connections. Therefore, a single "node" is represented as a server and its corresponding client. The "server" terminology used in the programmatic model for "client-server" architectures (a service requester and a service provider respectively) is different from the "server" used to describe a "node" in a network of devices used for the distributed system, as they had been used interchangeably in the other parts of this paper. For the remainder of only this section, the term "server" will be used to describe the service provider component of the "client-server" model.

The structure of the node is based on the client-server networking architecture. Each node is connected peer-to-peer with all other servers. Therefore, each node is able to communicate with all other nodes on the network, as they know the topology of the entire network. Each node consists of a server and a client. The server is able to receive updates only from its own client. The server can send updates only to other servers' clients. However, this split architecture is abstracted as one contiguous entity. This abstraction creates the illusion that nodes are able to talk to each other, when in actuality only the servers can talk to

only the client of other nodes, which in turn communicate each message to its own host server, as depicted in Fig. 1. Each server contains its own replicated database, creating a global RAID-10 model, in which a change in one database prompts a change in all databases.

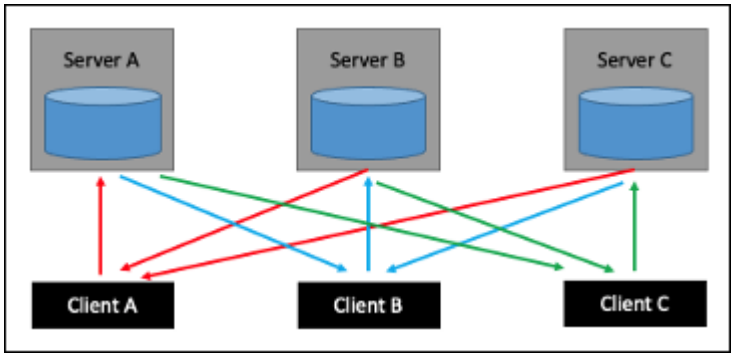

Fig. 1. Client-Server architecture for multiple node interaction.

In this peer-to-peer model, every server should be able to communicate with every other server either directly, or indirectly through an intermediate server. Servers have the ability to route traffic as a proxy on behalf of other servers if they are unable to reach an intended destination. Under this model, it is also assumed that in the absence of partitions, all communications are successfully received by all servers. That is, unless there is a fault within a server, or a partition among the servers, any transactions sent will be received for working servers, so resends are not necessary. However, transactions sent in a particular order may be received in a different order. Additionally, the proposal in this thesis allows the servers to enter and leave a network at any time, even during a transaction or the coordination protocol.

4.2.2 Design of Database Storage

The data storage system is designed as a relational database, but is fixed in that it only has 6 columns to store all necessary information for client usage and metadata for internal server operations: 1) PID - a hexadecimal number generated for identification purposes.

Every unique transaction is assigned a distinct PID, not just for unique entries, so that updates containing the same key-value pair are distinguished from existing data entries. 2) KEY - a key of any entry, stored as a string. Each key may not necessarily be unique. Confirmed and consistent keys with a consistency status of "0" will be unique, but keys with different statuses may exist. 3) VAL - a value entry associated with a key, stored as a string. May be duplicated as its entire environment is in relation to its key. 4) CONSISTENT - an integer value ranging from -4 to 4, indicating the consistency status for the data entry. 5) TIMESTAMP - an integer value representing the Unix epoch time that a single transaction occurred, provided by the timestamp issuing server. 6) PRIORITY ID - an integer value corresponding to the priority assigned to the transaction for conflict resolution tie-breaking purposes. The value is unique within the same timestamp. If undefined by the client, then it is set as the index of the thread on the timestamp issuing server that processed the transaction by the timestamp providing server. The usage of all six of these columns is shown in Table 1.

Table 1
Database Display of Sample Transaction Entries

PID	KEY	VALUE	CONSISTENT	TIMESTAMP	PRIORITY ID
4902881e1f3d9fac	x	4	0	72819283	3
d90b7c3cf291ad48	y	-76	0	72819286	1
0f288d3c21fbb1da	z	12	0	72819295	1

4.2.3 Data Operation Processing

There are only 4 operations allowing the client to interface with the database: 1) ADD [key] [val] - takes one key and one value as parameters and stores the entire transaction in the database if the key does not exist in the database. If the key already exists as committed, then

the key-value pair will override the existing entry, as an entirely new transaction. 2) DEL [key] - takes one key as a parameter and deletes the entire key-value entry if it exists in the database. Until the transaction is globally committed and confirmed, the consistency indicator will display "-1" regardless of the partition status of the network, in order to allow the database to rollback any data if needed. 3) GET [key] - takes one key parameter and returns the corresponding value if it exists in the database. If multiple entries of the key exist, then it will return the value corresponding to the most consistent key. 4) PRT - displays a text output of all of the entries in the database.

The first three of these commands provide full read, write, and delete properties necessary to modify a database. The ADD and DEL operations are dynamic, while the GET operation is static. Dynamic operations enact a change in the contents of the databases and depending on the policy of the distributed system, require coordination to inform the other nodes of a transaction. Any update of a key-value pair into a single database is conducted atomically, but there is no guarantee that updating all databases is atomic, hence the need for a consistency indicator. Static operations do not change the contents of the database and are rather used as diagnostic tools that allow the client to interface with the database by displaying the key-value pairs of the database. In order to demonstrate the involvement of network functions to process the updates, Fig. 2 depicts the communication protocol necessary to confirm a single transaction in an ideal case.

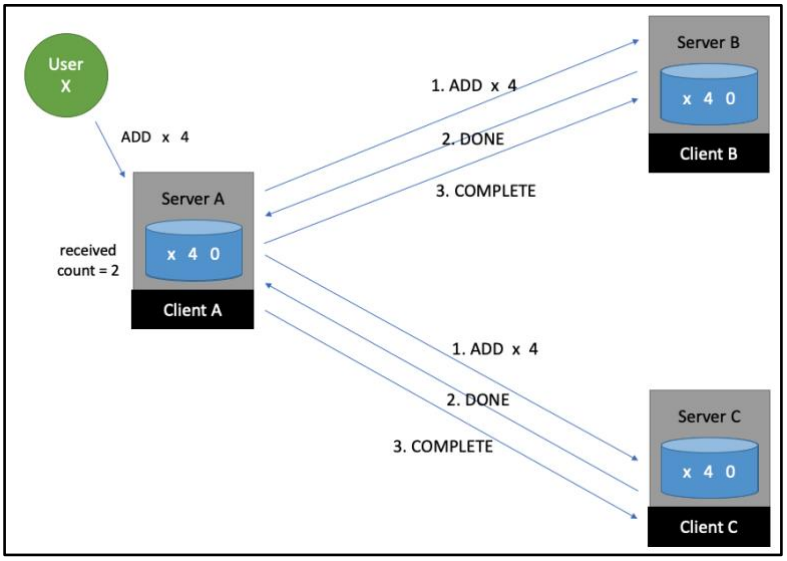

Fig. 2. Transaction propagation protocol between servers.

As can be seen in Fig. 2, each database stores items sequentially based on temporal locality of requests. This means that the server immediately appends any new and valid key-value pair into the database before propagating the update to the other nodes in the network. The receptor nodes validate the transaction before appending it to their databases. The order that a node receives transactions from users and other nodes may be different for each node. As a result, sorting the database is unnecessary, as it would generate a different sequence of key-value pairs for different databases. Valid dynamic transactions in the distributed system always update the database regardless of their commitment by other servers because the consistency indicator allows the clients and the servers to identify the status of each key-value pair. This is particularly necessary for DEL operations that are not committed, because if the entry is deleted entirely, then there is a significant overhead in having to rewrite the transaction. Instead, by modifying the consistency indicator to the appropriate status, any uncommitted DEL operations can easily rollback to the previous state. Additionally, any

subsequent ADD operation on the same key should override the DEL operation. The priorities of different transactions and their effects are detailed in section 5.3.

If the distributed storage system is constructed with the specified architecture, storage capabilities, and communication protocols, then the following proposed methods will provide verifiable consistency to the clients.

5 PROPOSED METHODS FOR VERIFIABLE CONSISTENCY

This section presents four different components necessary to provide a verifiable consistency: a universal timestamp server, consistency status indicators, a conflict resolution protocol, and a distributed coordination protocol. The proposed four components must be mutually utilized to achieve a fault tolerant and trusted distributed system. The following subsections will describe each method in detail.

5.1 Universal Timestamp Server

The universal timestamp server is a server that issues timestamps for global transaction comparison. Leslie Lamport's paper on distributed time sequencing showed how it's possible to map a partial order of events in a distributed system, but also clearly demonstrated that without having a synchronized clock it is impossible to achieve a global ordering. However, a synchronized distributed clock is infeasible for reasons described in section 2.1. Therefore, in order to get all databases to agree on transaction sequence/ordering, there must be a single universal trusted timestamp server that issues timestamps for a transaction and the corresponding servers. This is opposed to having each node depend on a physical time clock. Upon the initial setup of the distributed system, all servers will participate in an election to determine which one will be granted the additional responsibility of certifying the time of each transaction.

5.1.1 Election of the Universal Timestamp Server

The election works by having each server in the system generate a random positive integer and sending that value to all other servers for comparison. Each of the servers will receive the random integers from their peers and store them in an array with the array index

uniquely corresponding to the index of a particular server. At the end of the voting period, all servers should have an identical array and will proceed to select the leader server by searching for the index of the minimum element in the array. The server that proposed the lowest unique integer will be selected as the leader. Since the underlying assumption is that all servers are trusted it can be safely assumed that no server deviates from this protocol to truly generate a random number and select the correct leader for the purposes of an expeditious election so that distributed transactions can be processed. In the unlikely event that no integer is selected uniquely, the process is repeated. At this point, the server that proposed the minimum value will assume the role as leader and start accepting transactions to be signed, while also handling its own client requests. Once the leader initializes and sends a non-blocking "heartbeat" to all other servers in the network, the servers can initiate the timing validation through the leader. Having a universal timestamp issuing server does not undermine the fact that all servers are trusted. Even though all servers can be trusted to report their own local transaction times correctly, the purpose of having such a server is to mitigate the effects of inconsistencies and uncertainty in synchronized local clocks, which is why only one server can be the sole arbiter of time. Since the single timestamp server must service signature requests from all other servers, it will undoubtedly experience high traffic. In order to produce high throughput, the server issuing the timestamp signatures should reserve 1 thread to handle requests from each client, n threads to handle communication from its n peers, and dedicate all remaining threads and computational resources to servicing the time sequencing requests, since all transactions are guaranteed to get this sequencing verified.

5.1.2 Time Validation

The validation of transaction times works as shown in Fig. 3. When a server receives a request from a user, for example the transaction "`ADD z 43`", it will send the transaction to the timestamp server. The timestamp server will return a data structure containing the transaction, the time in milliseconds from Unix epoch, and the integer value of the relative priority of the transaction. The first component is necessary for the requesting server to distinguish between different transactions that it sent for timestamping. The second component is the global time issued in order to compare times with a discrete monotonic counter. The third component is a tiebreaker in order to assign the priority to the higher priority ID value in the event that two transactions were processed by different threads at the same epoch time. Since the system is to be trusted, there is no necessity in having the transaction cryptographically signed, as the intent of the initiating server is to obtain an unambiguous global time. This is sufficient for the server to send this transaction along with the timestamp to the other peers in the network for commitment. For untrusted distributed systems, a cryptographic signature in which the timestamp server disseminates a public key would be necessary to prevent tampering and to have all peers commit a transaction. Fig. 3 below demonstrates how a regular server first gets a transaction timestamped by the designated timestamp server before propagating it to peers. Server A initially sends the requested transaction to the timestamp server E, before receiving the time information and disseminating the result to all other servers.

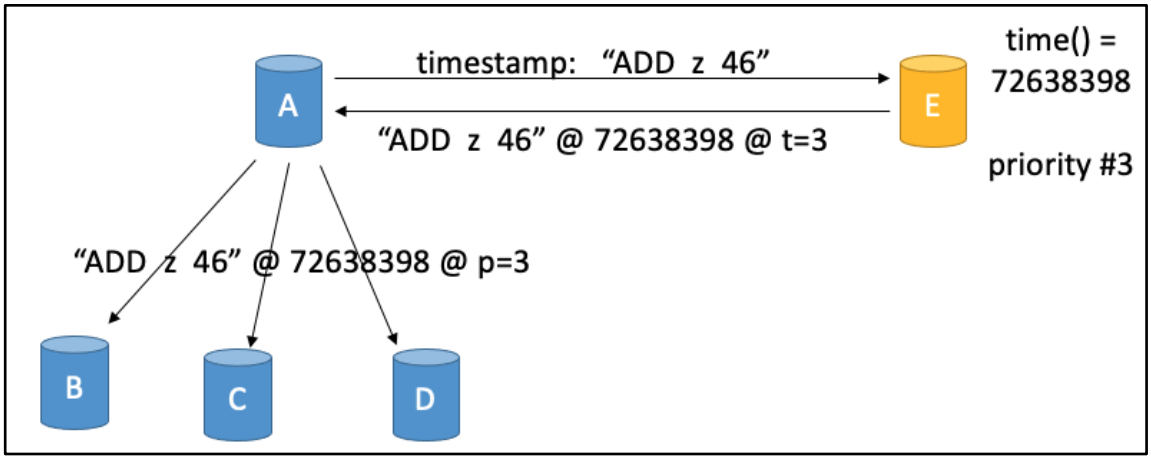

Fig. 3. Timestamp validation protocol.

5.1.3 Leader Fault Protocol

At some reasonable and predetermined time interval (10ms, 5s, etc.), the timestamp server should consistently send a heartbeat message to all other servers to indicate that it is still active. If any recipient server misses three heartbeat messages in a row but is still connected to other servers, then it can initiate the election phase by proposing a randomly generated integer. If it receives no integer from other servers, then it is possible that this server is the only one that is disconnected from the timestamp server. If the server cannot communicate with any other servers, then it will recognize that it is partitioned. If the server can communicate with another server that can communicate with the timestamp server, then the other server will relay time signatures to the timestamp server for it. If the other contacted server is also unable to connect with any other server, then both are partitioned together, and they will conduct a local election to service transactions from clients while being in a partitioned yet available state. Detailed descriptions and protocols for partitioned states are described in section 5.2.1.

In the case of any discrepancies or conflicts with distributed database transactions, a server can compare the timestamps of received transactions to assign priorities to later transactions. After a server receives a transaction with a verified timestamp, its database stores the value with the data structure fields received in the timestamp. At a later time, if a transaction is received that has an earlier timestamp, then the database rejects the least recent timestamp since it's no longer valid, favoring the latest timestamp. This demonstrates the utility of having a universal timestamp and how it can improve parsing for valid distributed transactions, while ensuring their sequential ordering.

5.2 Partition Consistency State Indicators

The consistency state indicator is a numerical value between -4 and 4 that represents the relative consistency of a data entry with regards to the other servers. Eric Brewer's CAP theorem demonstrated that if a distributed system is to exhibit partition tolerance, then it can either only be available in its ability to service requests from clients, or consistent in the state of replicated data across all servers. This is intuitively true because a network partition prevents different servers from communicating with each other, so if the distributed system is in a state of consistency before the partition, then by making itself available, it is subject to the unvalidated and uncommitted updated transactions. Otherwise, in order to maintain its consistent state, it cannot allow any uncommitted updates until the partition heals. The rest of this section will consider a system that remains available during partitions, and in order to improve database coherence after the partition, this thesis presents a protocol to achieve partition consistency to assign quantifiable priority metrics to the post-partition conflict resolution process.

5.2.1 Partitions Classification and Detecting Partitions

In order to proceed with the aforementioned protocol, it is first necessary for any server to detect any faults or partitions. A fault or a partition causes the distributed system to behave in an unintended way, which necessitates a change in the protocol and behavior of the other servers to accommodate for the unintended behavior. Therefore, when the distributed system is operational without partitions, all servers need to poll all other servers at regular intervals to ensure that they are active [29]. This will allow servers to detect network partitions or detachments and modify their protocols to incorporate the partitioned state, since it is assumed that this peer-to-peer system is a mesh network. For the remainder of this section, the term “server” and its equivalent in the topological context, “node”, will be used interchangeably. There are 6 statuses of faulty or partitioned servers defined as such:
1) Entirely Isolated Inactive Server: When a single node detects no other nodes or is unable to communicate with the user(s), it is "down" regardless of whether the node is functionally operational or terminated by fault as the result of a program crash. As a result, it is unable to participate in any capacity as an active partitioned node, an available read-only database, or a replicated state machine. 2) Non-interfaceable Active Replicated Server: If a node is connected to all of the other nodes of the network but not to any user, then it exhibits an active replica state, since it cannot accept new updates from users and therefore cannot propose anything. It is just used as a backup data store and actively processes transactions sent by other servers. 3) Isolated Partitioned Server: A node that is only connected to the user(s) and able to process updates locally, but unable to communicate with any other servers in the network. It cannot receive reads, writes, updates, or status messages with other servers,

but remains available for local updates to any clients that interface with it. 4) Read-only Inactive Replica Server: If a node is able to receive but not send, it definitely does not support consistent writes. It can accept them, but label them as dirty. then depending on the architectural model, it may be read-only, if the model requires confirmations back to the proposal node. Otherwise, it can simply accept updates and label them as dirty. 5) Write-only Inactive Replica Server: If a node is able to send but not receive, then it would not know that other nodes can hear it and thereby marks itself as "down". Because the node would not be able to receive heartbeat messages, it would assume that it is functional while all other nodes are not functional and mark itself as a singularly partitioned node. 6) Active Partitioned Server: If a node is connected to the client(s) and only a subset of other nodes in the system while being able to read and write, then it is a part of a partition.

In the event of no partitions, consensus and coordination will provide consistency of data across all servers while connectivity to the clients will provide availability of data access on any server. In the event of partitions, the system must recognize each case as defined in this section and adapt accordingly. By polling all servers and modularly testing the working functionality of the components of the server, each server will be able to determine not only if it has faulted or partitioned, but also if another node has faulted so that it can recompute its topology to accommodate the distributed system and update the consistency status of transactions accordingly in order to achieve verifiable consistency for the post-partition conflict resolution.

5.2.2 Consistency Indicator Bit Protocol

Any subset of servers that is isolated by the partition must functionally operate as if it is connected to the entire graph in order to achieve consistency across the partitioned nodes. This means it must perform timestamp server elections and send transaction updates behaving as if all other nodes are in contact but knowing that it truly is in a partition. Otherwise, any server that propagates a transaction will keep waiting for nodes in a different partition and receiving nothing back, causing every update to be canceled thereby hindering any progress. First and foremost, any entries in the key-value store database that have a consistency status indicator of "0" all must now be changed to "3" to indicate that this key-value pair was previously consistent when the partition did not occur and that there is no guarantee that it will remain consistent throughout the course of the partition. Any subsequent updates and transactions for the duration of the partition can never be marked with a consistency status of "0" or "3". However, any subsequent partitions do not need to be tracked.

5.2.3 Processing ADD Transactions

The ADD operation is a dynamic operation that appends a key-value pair entry to all databases in the distributed system by creating a new row. Any new ADD transaction proposed by a source server will initially be stored as an uncommitted entry in its own database by marking the entry with the consistency bit of "1" to indicate that the transaction has not been committed in any other servers' database. If no other servers exist within the partition, then all updates will receive the bit "1" to indicate that they are not replicated and always remain inconsistent. When the ADD transaction is propagated to the other accessible

servers within the partition, the source server coordinates a confirmation protocol to ensure that all servers within the partition have adopted the transaction. Upon confirmation, the source server relays a “COMPLETE” message to the peers, for which all servers will change the consistency bit to “2” indicating that the transaction, and its corresponding entry in the database, is consistent among the nodes in the partition. If some recipient server faults before changing the entry’s consistency bit to “2”, the protocol will still be valid since the faulted node behaves as though it is in a partition of itself. Fig. 4 depicts the relative consistency of partitioned nodes and their usage of consistency status indicators.

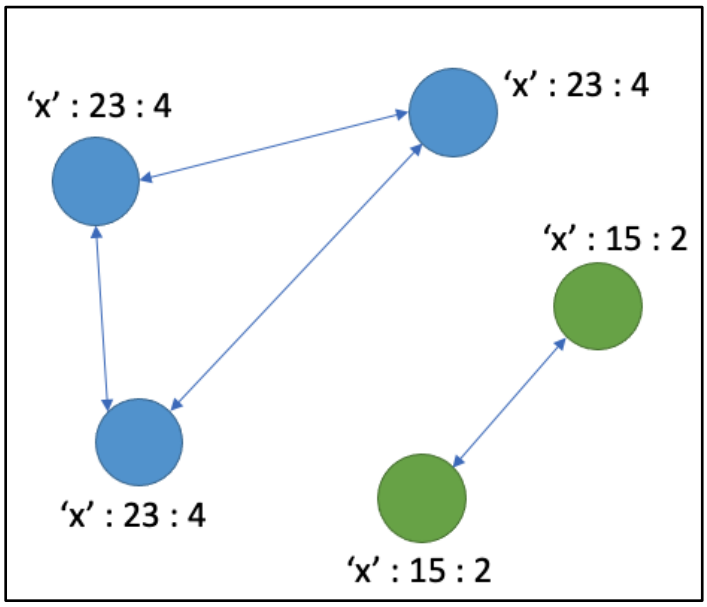

Fig. 4. Local consistency of transaction entries within 2 distinct partitions.

5.2.4 Processing DEL Transactions

DEL transactions must be conducted in a different manner than ADD transactions due to the deletion of data. In a non-partitioned distributed system, when a deletion is requested for a particular entry, the goal of the system is to have the entry permanently deleted from all databases and confirmed as consistent across all databases. However, for a partitioned

system, it is necessary to have a rollback mechanism for a database entry in case servers on one side of a partition request to delete an entry, but servers on another side of the partition actively and frequently use the entry. During the conflict resolution stage, priority would be given to the least recently used transaction, the key updates. In order to prevent the overhead of identifying all of the nodes that deleted the specific entry and retransmitting the transaction data, it will be significantly more efficient to designate a consistency status of "-1" to indicate DEL operation on an unconfirmed key-value pair. Inside of a partition, if all nodes confirm a DEL operation, they can change the consistency status to "-2" indicating that the transaction has achieved local consistency. If this is achieved, the databases will ignore the entry for subsequent ADD and GET operations, allowing conflicting key-value pairs to be added. When there is no partition in the system, upon the globally consistent confirmation of a delete transaction, all "-1" statuses will be simply deleted permanently along with the transaction. At this point no backup is needed for the transaction for rollback purposes because the delete operation has been verifiably consistent. Table 2 below lists all of the possible consistency statuses with their use cases.

Table 2
Description of Consistency Indicator Values

Consistency Indicator Value	Relative Consistency Status
-4	consistent DEL transaction within a quorum
-3	previously globally consistent DEL transaction
-2	locally consistent DEL transaction across partition
-1	inconsistent or unconfirmed DEL transaction
0	globally consistent entry across all servers
1	inconsistent or unconfirmed ADD transaction
2	locally consistent ADD transaction across partition
3	previously globally consistent ADD transaction
4	consistent ADD transaction within a quorum

5.2.5 Quorums and Consistency Bit Update Protocols

If a group of servers writes the value within a partition is able to achieve local consistency and a confirmed key-value pair, the consistency bit for that entry across all databases is verifiably a “2”. When partitions occur, servers may be a part of the majority of a group of nodes. For example, in a topology of 10 nodes, if 3 nodes are partitioned from the other 7, then the 7 nodes will be designated as a quorum. If a group forms a quorum, then the consistency bit for any ADD transaction processed by this group will be a “4”, while the consistency bit for a DEL transaction processed by this group will be a “-4”. If a server is a part of a quorum but still in a partition, then in the event of disputes upon post-partition

conflict resolution, its transaction updates take a higher precedence to all other servers in the partition not in the quorum.

5.2.6 Functionality of the Consistency Bit Update Protocol

This philosophy behind adding consistency indicator bits to transactions in the database is so that it modifies the data so that it reflects uncertainty, while providing accurate status information. In the event of inconsistency, if a client needs data available, they will always know that the data is inconsistent globally and can choose whether or not to wait for convergence and global consistency. Consequently, this protocol will significantly simplify the process of comparing priorities and recency needed to achieve consensus across all servers upon being available after a partition. The implementation of a consistency bit further assists the consistency comparison of distributed. Additionally, the updates to the transactions required by including an indicator bit preserve the ACID guarantees of databases. The initiator node, the server receiving queries from the user and sending to recipient nodes, acts as the system coordinator for the duration of the transaction commitment. An ADD transaction, for example, sent by the coordinator has an implicit dirty bit of "1" sent to all databases, regardless of the partition status. If the communication fails upon initial sending of the transaction, then the transaction is never sent, thereby satisfying atomicity. Because each transaction is processed with reference to the transaction ID, the unpredictable acts of context switching, multithreading, and network faults will not affect how the transactions are processed and stored in the databases. Every action that a distributed system or the database of a server takes is in relation to the transaction number, a primary identifier of the transaction, so there is no way for multiple transactions to be mixed up, or

for any operation to result in an inconsistent state. These two properties satisfy the consistency and isolation criteria for ACID databases. Finally, in the presented protocol, durability is bypassed locally for the sake of global consistency, which significantly improves non-volatile storage overhead during partitions. For example, if the system is not partitioned, then every transaction can be confirmed, and all confirmed states must be persistent. Fundamentally, during a partition, since consistency is not guaranteed, there is no need to ensure the durability of non-confirmed transactions if the consistency bit is not "2".

While the use of a "dirty bit" has been extensively used in database management with regards to page table entries, caching, and record index mapping, the complex conditions involving various types of partition cases require a more nuanced definition of partition cases and an indication of consistency in each case. Therefore, it was necessary to determine an optimal heuristic for how different transactions are committed by different subsets of nodes. The consistency indicator is never used before or during a partition because it is meant to indicate the state of the servers in a partition. However, the indicator will be extremely useful for post-partition conflict resolution, specifically when assigning priorities to conflicting entries of different partitions. The values of the indicators will help any conflict resolution algorithm determine whether nodes were in a quorum, have confirmed and consistent transactions, among various other statuses.

5.3 Post-Partition Conflict Resolution Algorithm

Maintaining data consistency across all of the servers is arguably the most important property of a highly available distributed system because it allows clients to update and use data with the confidence that the data is accurate, persistent, and replicated. Therefore, after a

partition, the distributed system must incorporate a protocol to identify all inconsistencies across the servers' databases, determine which key-value pairs have priority over others, and issue a correction such that all nodes eventually and expeditiously update their databases to be identical to each other. At the same time, servers must respond to requests made by clients, elect the timestamp server, and manage the polling and fault detection of the system. Traditionally, the consensus process is performed by electing a leader node that unilaterally selects which data to include in the final replicated state. This may be done without regards to operation type or recency of transactions. However, with the new information that the timestamp server and the consistency status indicator provide, namely verifiable consistency on a local level and global time sequencing, the proposal of a post-partition conflict resolution algorithm takes into consideration all 3 criteria in addition to a practical user-oriented perspective when making decisions about which entries to keep and delete from the union of all entries across all databases. The rest of this section will consider a distributed system in which the nodes have reconnected after a partition.

5.3.1 Consideration of Mid-Partition Availability

Consider a simple distributed system of 3 nodes that experience updates during the system partition. Fig. 5 below depicts a discrete time event map of the connectivity of different servers. This is useful to demonstrate each server's role in a partition and how that affects consistency. Each data entry consists of a [key : value : consistency status] tuple. The following figure depicts the relationship between the nodes over time and the updates processed by each as partitions occur.

time	Node A	Node B	Node C
0	*<All nodes are connected.>*		
1	‘x’ : 56 : 0 ‘y’ : 78 : 0	‘x’ : 56 : 0 ‘y’ : 78 : 0	‘x’ : 56 : 0 ‘y’ : 78 : 0
2	*<Node C gets disconnected from Nodes A and B.>*		
3	‘x’ : 56 : 3 ‘y’ : 89 : 2	‘x’ : 56 : 3 ‘y’ : 89 : 2	‘x’ : 56 : -1 ‘y’ : 78 : 3 ‘z’ : 96 : 1
4	*<Node B disconnects from Node A. All nodes are disconnected from each other.>*		
5	‘x’ : 56 : 3 ‘y’ : 89 : -1	‘x’ : 56 : 3 ‘y’ : 93 : 1	‘x’ : 56 : -1 ‘y’ : 84 : 1 ‘z’ : 96 : 1
6	*<Network heals. All nodes are connected.>*		
7	?	?	?

Fig. 5. Time diagram of key-values pairs for each server.

Initially all of the nodes are connected and as a result, there is no issue regarding the consistency of the data that is written to any single node because an immediate update will occur to all the other nodes. This becomes verifiably consistent when the consistency indicator for that key-value pair becomes “0”. Then at time T=2, Server C becomes disconnected from servers A and B, who are still able to communicate with each other. All previously consistent key-value pairs must update their consistency status from “0” to “3”. At time T=3, servers A and B update the key ‘y’ to have the value 89, and a consistency status of “2”, since this is locally consistent among 2 servers. Upon post-partition convergence, the servers will identify the connection between servers A and B as a quorum, but this is known only to servers A and B during the partition. This may not always be the case during a partition. Concurrently, Server C deletes key ‘x’, which mandates a consistency bit of “-1”. It does not receive any updates to key ‘y’ but is requested to add a new key-value pair of (‘z’ : 96), which remains local. At time T=4, Server B disconnects from Server A, thereby leaving all servers partitioned from each other. At time T=5, each server makes its own update. Server A deletes the key ‘y’, Server B updates the value of key ‘y’ to be 93, and Server C updates the value of key ‘y’ to be 84. As servers remain available to their respective clients, the inconsistencies from the previous point of consistency grows for every unconfirmed transaction. As a result, at time T=7 after all the servers become reconnected, there needs to be an appropriate algorithm to provide consensus by assigning priorities that take into consideration updates that were made by quorums, recency, and operations, all throughout the partition.

5.3.2 Algorithm

In order to achieve the most fair and optimal collection of entries, every node must consider the timely sequence of transactions across all nodes. This requires total coordination between every single node and its entire transaction history. Sharing such a history allows the other nodes to uniformly determine which nodes were a part of a quorum, and which transactions were confirmed as consistent within a partition, giving priority to those transactions over transactions that were never shared with another node or other locally unconfirmed transactions. Globally, priority is assigned as follows: 1) transactions that were completed recently as opposed to transactions that were completed previously. Any conflicts are broken by assigning the value to the most recently achieved consistent state within a quorum or a locally consistent subgraph. 2) transactions that have confirmed consistency within a quorum partition. 3) transactions that have confirmed consistency within a non-quorum partition. 4) ADD operations over DEL operations to prevent the loss of data by any party. 5) Priority ID for processing conflicting updates that were made to the same key at the same time on different partitioned machines. The post-partition conflict resolution algorithm works as follows:

1) Each server receives the full transaction history for all other servers in the network.

2) Each server generates a list of all keys assigned across all servers, the "key_superset".

3) For each key in the key_superset, a server searches for the most recent transaction involving the key across all transaction histories. Exactly one transaction for each key can be included in the final state of the replicated database.

4) If multiple transactions involving a single key occur at the exact same timestamp, then the final result is selected based on following hierarchy:

a) quorum consistency - transactions that are consistent within a quorum.

b) non-quorum local consistency - transactions that are consistent within a subset of nodes that do not make up a quorum.

c) ADD operations over DEL operations - transactions of the DEL operation will be rejected in favor of those consisting of an ADD operation.

d) Priority ID - higher value of Priority ID is preferred.

5) The final set of key-value pairs is determined and compared with all of the other nodes for consistency. If consistency is confirmed, then the bit of '0' is stored.

Step 1 and step 5 are the only steps that involve coordination of all of the nodes. All other steps are performed by autonomous calculation to produce a result. The following pseudocode in Fig. 6 demonstrates how the algorithm may be implemented in code:

Algorithm 1: Post-Partition Conflict Resolution Algorithm

Data: superset of all transactions across all available servers
Result: valid key-value entries after the partition merges
keyset ← *superset of all keys*;

pid ← *priority value for transaction*;
for $k \in keyset$ **do**
  $TX = \text{argmax}(\text{time}(T_{1k}), time(T_{2k}), time(T_{3k}))$
  if *TX has more than 1 entry* **then**
    for $txn \in TX$ **do**
      if *is_quorum(txn) X* **then**
        return txn;
      end
      if *is_minority(txn) X* **then**
        return txn;
      end
    end
    return argmax(pid(txn));
  end
end

Fig. 6. Pseudocode of conflict resolution algorithm.

By applying the algorithm described above to the time graph of the distributed storage system depicted in Fig. 5, a globally consistent and replicated state of data can be achieved. The most recent update of key ‘x’ is a DEL operation that occurs at timestep T=3 on Server C, while there are no updates to key ‘x’ anywhere else. Therefore, the final state of the database must not contain ‘x’. The most recent update of key ‘y’ occurs at timestep T=5 on all 3 servers, which involve a DEL operation and two ADD operations of different values. Because no quorum was established and ADD operations are prioritized over DEL operations, the tie is broken by the transaction with the higher priority value. Since this is not specified in the diagram, for the sake of this example, this paper will arbitrarily assume that the transaction into node B has a higher ID although in the real implementation, all values

will be presented for evaluation. Finally, the most recent update to key ‘z’ occurs at timestep T=3 as an ADD operation. Therefore, the final state of the database will contain the key-value pairs: (‘y’ : 93) and (‘z’ : 96).

The ultimate goal of this conflict resolution protocol is to rely on programmability and algorithms to resolve issues of conflicting key-value pairs when they occur instead of delegating this responsibility to the user that may take hours to comb through conflicting entries, or a unilateral elected node that may make arbitrary decisions whose internal processes cannot be validated by its peers. Additionally, if the leader faults, then this creates significant delays in resending all of the transaction history to a newly elected leader due to the transmission and election overhead. Therefore, by having all processes autonomously and deterministically decide the correct output of a sequence of instruction servers as the best option to achieve data consensus. Based on the architectural assumptions presented in section 4.1, it can safely be assumed that no server maliciously falsifies information and if each server correctly follows the algorithm, then they will all produce the same output. Since all servers must collaborate to validate the final result to denote a consistency indicator of “0” for all data entries, in the event that a server faults and produces an incorrect result, the servers will just take the majority decision, relying on the probabilistic assumption that the probability of a fault at any given time is very unlikely. Therefore, regardless of the circumstance, this algorithm will allow the distributed storage system to produce a reasonable resolution to any conflict that may arise in the partition.

5.4 Distributed Transaction Coordination Protocol

Managing how distributed transactions update the databases of a distributed storage system is a fundamental functionality of any distributed application. While the underlying protocol enabling the consistency status indicator provides verifiable consistency, this property is only possible if it operates using a communication protocol that ensures that servers have received updates and actively change their databases to reflect the updates. Using the newly proposed universal timestamp server and consistency indicator protocol as a basis, this thesis presents a concurrency-focused distributed transaction protocol.

5.4.1 Managing Concurrent Distributed Transactions

Consider a simple distributed storage system with 4 servers labeled A, B, C, D, such that servers A and B are close to each other while servers C and D are close to each other, such that the two sets of servers are far apart. All of the servers are working perfectly in coordination and currently have the key-value pair (‘y’ : 1) replicated consistently across all machines. Additionally, each data entry is tagged with the consistency status bit ‘0’ since this transaction has been confirmed to be consistent. Later, at the exact same time T, machines A and D independently receive requests from their respective clients to add a new key, namely ‘x’ with different values. Server A assigns the value ‘2’, while Server D simultaneously assigns the value ‘3’. In their internal databases, the transaction is entered as an entry, but the consistency bit is set to ‘1’ for both nodes (A and D) since the update is not yet confirmed among all servers. As a result, Server A propagates the new update to servers B, C, and D. Concurrently, Server D propagates its new update to the rest of the servers as well. Both source servers then wait for confirmation from all 3 receiving servers. Due to its proximity to

Server A, Server B receives the update of key ‘x’ almost instantaneously. Since the transaction contains a new key-value pair and does not conflict with any existing entries, server B will accept the transaction and store it in its local database with a consistency bit of “1”, thereby sending a “DONE” message to Server A. Server B waits in the background for confirmation from Server A that this transaction is ready to be confirmed. Later, Server B receives the update of ‘x’ from Server D, but recognizes that it has already accepted an update for the same key from server A. In order to prevent conflicting transactions from being committed, it first checks the priority ID for the transaction. As defined in section 5.1, the timestamp server issues a priority ID number, essentially the thread number to break ties in the event that two transactions have identical timestamps, as in the case of servers A and D. For this example, Server D’s transaction has a higher priority than server A’s transaction. Therefore, Server B will also accept Server D’s transaction. In fact, Server B will accept any subsequent transaction for a conflicting key ‘x’ that occurred at the same time if it has a higher priority value, without needing to retroactively reject a previously accepted but lower priority value. All servers must accept any previously unseen key-value pair transactions, reject any subsequent key-value pair transactions with a lower priority, and accept any key-value pair transactions with a higher priority. This scheme will always work for a trusted distributed system because it is guaranteed to have at least one server reject a transaction with a lower priority ID.

Concurrently, and due to proximity, Server C will receive the transaction from Server D first and since this is a new and non-conflicting transaction, it accepts the transaction, stores it in its database and sends a “DONE” message to Server D, just as Server B did with

Server A. However, Server C eventually receives Server A's transaction proposal of its key-value pair and notices that there exists a conflict in its database as it has the same key that it received from Server D. After checking the priority ID of the transaction, Server C sends a "REJECT-DUP" message to Server A effectively rejecting the transaction from Server A. All servers that receive a subsequent conflicting transaction with a lower priority must explicitly reject the transaction in favor of the higher priority transaction.

Finally, Server A and Server D must eventually send their initial proposals to each other. Even though Server A has already entered the value '2' into its database for the key 'x' and received the transaction confirmation from Server B, it must acknowledge the rejection it received from Server C and must accept the higher priority transaction from Server D. Then Server A will update its local key-value pair to the one sent by Server D while rejecting its own entry for key 'x'. Since Server D recognized that its transaction has a higher priority than Server A, it effectively rejects Server A's transaction and as a result, Server A does not have the necessary confirmations from all of the recipient servers to send a "COMPLETE" message to all of the servers. In contrast, Server D with a higher priority is able to get all of its transaction proposals approved and is therefore able to confirm the consistency of the proposed key-value pair of the transaction across all databases in the distributed system. At this point once all databases change their consistency bit to "0", consistency is verified for that entry. The entire protocol is depicted in Fig. 7 below.

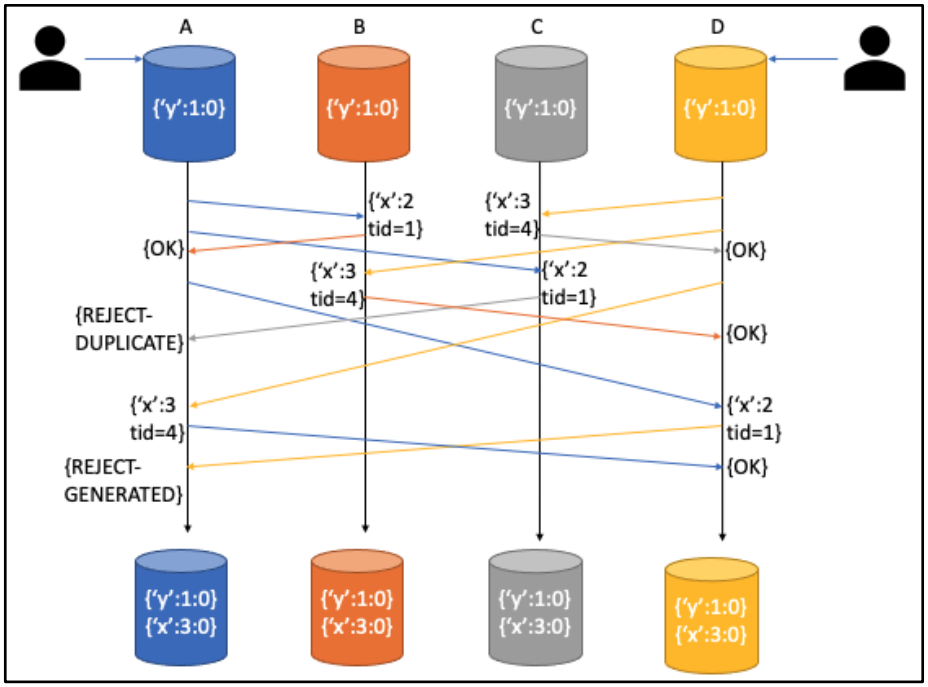

Fig. 7. Concurrent conflicting transaction protocol.

5.4.2 Canceling and Rejecting Messages

Continuing with the same scenario as above, Server A does not need to send a cancellation message to the other servers in the system because Server A will never confirm the consistency of the key-value pair to all servers. However, for systems with low traffic but severe database delay and overhead, it may be useful to implement such a cancellation alert to free up memory as soon as possible. This additional requirement would only be useful for efficiency and optimization purposes, not functional purposes. Additionally, by accepting the transaction with the higher priority, the recipient servers are inherently rejecting all previously accepted transactions with a lower priority. It is impossible for Server A to send a "COMPLETE" message to all servers because when Server D receives the transaction proposal from Server A, it will send a "REJECT-GEN" message to Server A, indicating that it, as a source node, proposes a conflicting transaction with a higher priority. By following

this process, all servers can ensure that in the event of conflicting entries, they will all select the entry with the higher priority ID and reject the lower priority transaction. For subsequent lower priority transactions that occur at the same time, namely the proposal from Server A, Server C will send a "REJECT-DUP" message to the source node (Server A). At this point, rejecting low priority transactions becomes monotonic because no further information can lead to a different result for Server A's proposal. On the other hand, verifying a message in general is non-monotonic because every single node must be queried to ensure validity.

5.4.3 Multithreading and Locking

In order to expedite the global confirmation of transactions, each transaction will be processed by a different thread on a server, so there needs to be a locking mechanism on the database to ensure that any server will not concurrently accept a transaction on one thread and accept a conflicting transaction with a lower thread priority on a different thread. For example, consider a case where, by chance, Server C receives two conflicting transaction proposals from Server A and Server D simultaneously on different threads. If these threads do not coordinate or reserve access to the database, then when accepting both transactions, each thread will potentially override the database of the other thread's transaction. Even though, by accepting both transactions, the threads do not violate the principle of rejecting the lower priority transaction because the transactions arrived at the same time, this needs to be avoided because the principle relies on eventual database sequentiality and the fact that some node must receive the lower priority transaction later and will be rejected. However, without locking, this is not guaranteed under the very unlikely scenario that servers A and D receive conflicting transaction requests at time T, but while Server D is still updating its own

database, on a different thread it receives and processes the transaction request from Server A. This violates the tie-breaking principle of accepting the transaction with the highest priority, as it would cause further problems down the road if servers B and C also accept both transactions. After receiving “DONE” messages from all recipient servers, source servers A and D would send “COMPLETE” messages to the system again. This would cause the two conflicting transactions to be committed by each server to the distributed database, which would not be allowed. Therefore, there must be some locking mechanism on the shared resource of the database. Multithreading would expedite the processing of the majority of cases in which the transactions are not conflicting and independent, while locking would prevent the edge cases of conflicting data.

5.4.4 Handling Faults and Incomplete Transactions

Because servers are able to self-identify faults, any functioning initiator server can assume that all recipient servers have updated the consistency bit for the transaction. However, in the event that some server has faulted before receiving an update, it is necessary to consider 2 cases. The first case is in which the initiator node faults before being able to confirm the reception of all proposed transactions. In this case, the transaction is never confirmed, and its consistency bit will remain a “1” on all involved servers. The transaction is considered to have failed. The second case occurs when a recipient node faults after accepting a transaction. This event will still provide verifiable consistency because the transaction will just not be confirmed and the servers will not receive a pulse from the faulted server, then understanding that a partition has occurred, and therefore will readjust the topology of the network.

For a failed transaction that does not have any conflicts, the initiator node will also delete the request or, in the optimal case, save it to cache and ask the user to try again. At this point, it can respond to the user that the query was unsuccessful. However, the goal is to minimize user interaction, as the DBMS needs to resolve issues autonomously. Therefore, upon receiving a "REJECT-DUP" signal or a fault, the initiator node can wait a prespecified period of time for the cause of the "REJECT-GEN" signal, namely the proposal from the other node. If the time is exceeded and they do not receive each other's original proposal, then they will call "CANCEL" on the nodes that they received "DONE" from. This is better than canceling immediately after any “REJECT” signal, because it allows the key-value update to have some replica even though it may be wrong. However, this does not matter because the client can see that the consistency bit is “1” and act accordingly.

If a key already exists in the database, and later multiple concurrent transactions are trying to update it, it is necessary to keep the same key as confirmed. The intermediate servers B and C must respond to the source servers A and D, because if they don't and just automatically accept the most recent update, then they have no way of determining the state of the entire system for that variable and cannot accurately display a consistency status bit. The recipient nodes must respond to the first update they receive, so that they can mark the state of the key and listen to any further updates regarding that key. This will work for sure, because if the system faults for some reason, such as if Server A sends the proposal to servers B and C, then faults before sending to Server D, then consensus will not be achieved. Therefore, it is impossible for any node to have a consistency status bit of ‘0’ for that key.

The respective initiator nodes may receive the original update proposal from the other node if there are no partitions or transmission faults.

6 EVALUATION OF PROPOSED METHODS

As described in section 4, the system architecture of a distributed storage system needed in order to implement the proposed methods from section 5 and ultimately allow verifiable consistency to be achieved in the presence of different distributed system events like faults, partitions, and normal uptime. However, the proposed methods do not improve the performance of the system as compared to other distributed system implementations, nor do they enhance the existing abilities of distributed systems in any way. Therefore, the purpose of the experimentation section system is to demonstrate that verifiable consistency can be achieved by the implementation proposed in this thesis and display a relative benchmark for the performance of the distributed system at different configurations of server-client pairs (nodes). The following experimentation has been run on a simulated environment for testing and metric comparison purposes. Attempts to reproduce this experiment may produce different results depending on the implementation of the distributed system, implementation of the algorithms, frequency of transactions, network bandwidth, system hardware, operating system, and system software capabilities.

6.1 Experimental Setup

In order to benchmark the relative performance and processing efficiency of the distributed system, it was necessary to utilize a processor with many threads, dependable networking capabilities, and low memory utilization. Therefore, each server-client pair, abstracted as a “node” in the distributed system ran on an AWS EC2 cloud service instance running Ubuntu 20.04. Only the network configurations were modified beyond the default settings to allow TCP and SSH connections from other nodes, while all other settings were

left unmodified. Each EC2 instance is provisioned with a public IPv4 address which was used for client-server communication between the nodes. 25 nodes were created, and all IP addresses were collected after instantiation and hardcoded into the server directory for each node in the distributed system. This allowed each node to communicate with each other on a peer-to-peer basis. Finally, a user channel was created allowing a user, represented in section 4 as a client, to connect to a node and request updates to data using the specified operations from section 4. Then, Python code was uploaded to run the socket communication protocols between the nodes. At this point, a fully functional distributed storage system of 25 nodes was implemented to completion and could service basic operations, elect a timestamp server, handle partitions, resolve conflicting transactions, and coordinate amongst the nodes to achieve verifiable consistency. Nodes could be added and removed by simply turning off EC2 instances and notifying the rest of the nodes the new size of the system. Experiments could now be set up to analyze the capabilities of each component. The user-to-node connections served as the interface for allowing each node to post its timing results after running tests.

6.2 Results Analysis and Discussion

In the subsequent sections, this thesis will describe the methodologies for conducting an experiment to measure an important metric based on each of the proposed methods from section 5.

6.2.1 Experimentation of Timestamp Server

Initially, the processing ability of the timestamp server was checked for correctness. As can be seen below in Fig. 8, Server B is able to connect with the elected timestamp server

and send an ADD transaction of “ADD x 75”. The timestamp server almost instantaneously returns the transaction along with the timestamp listed as the Unix epoch time in milliseconds and the priority ID value, defaulted as the ID of the thread that serviced the transaction.

```
...ServerB connected to Time Oracle Server...
ServerB: ADD x 75
ServerB: Time Oracle: "ADD x 75" @ 16480760750674272 @ 36

ServerB: ADD y 23
ServerB: Time Oracle: "ADD y 23" @ 16480760893674660 @ 36
```

Fig. 8. Response from timestamp server to server proposing transaction.

In order to further analyze the transaction processing performance of the timestamp server, an experiment was set up to measure how many transactions could be processed. Each server was set up to constantly send as many transactions as it could until it was stopped by the timestamp server. Consequently, the timestamp server was tasked with processing and signing as many transactions as it could for a prespecified number of seconds. This experiment does not make any claims on the efficiency of the timestamp server in the normal use of a distributed system, as a distributed system will likely be faster without such a bottleneck. Instead, this experiment displays the relative computational output based on the number of nodes in the system. Fig. 9 depicts the average number of transactions signed and returned by the timestamp server for each time interval. Each data point has been calculated by averaging 10 sample points.

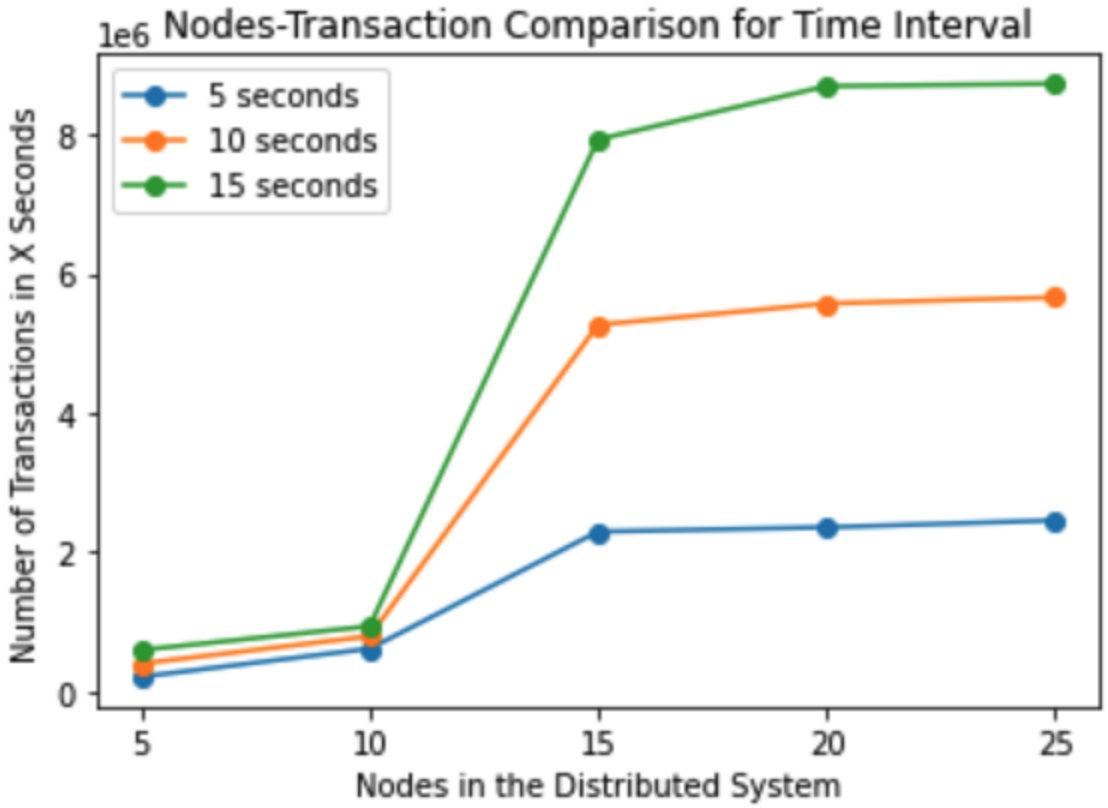

Fig. 9. Performance of timestamp server.

As can be clearly seen by the graph, for up to 10 servers in the distributed system, the number of transactions able to be processed by the timestamp server is very similar to that of other nodes regardless of the time allotted. However, after the number of servers exceeds 10, the differences in the number of transactions processed is extremely distinct. There is a drastic shift in the number of transactions processed from 10 to 15 servers. For 15, 20, and 25 nodes in the system, the growth of transactions processed is very minimal as the value tends to converge. This indicates that as the number of servers in the system grows beyond 25 servers, the number of transactions in millions able to be processed by the timestamp server will grow slowly.

6.2.2 Experimentation of Consistency Status Indicator

In addition to testing the performance of individual components, it is necessary to demonstrate that the distributed system with the proposed methods implemented is behaving correctly and is able to verify the consistency of data by using the consistency status indicator. Therefore, a test was set up on the user end to include a user interface that would allow the user to see the status of data transactions in the system during a consistent time and during a partition. Fig. 10 below shows a user that logs into Server A. Then the user runs two ADD transactions to the database: "ADD x 78" and "ADD y 34". Immediately after printing the constructed database, by viewing the consistency bit of "0" for both entries, it is clear that across the entire distributed system, both key-value pairs have been propagated and confirmed. Similarly, after running the transaction "DEL y" and printing the database, the lack of the key "y" in the database reveals that it also has been deleted across all nodes.

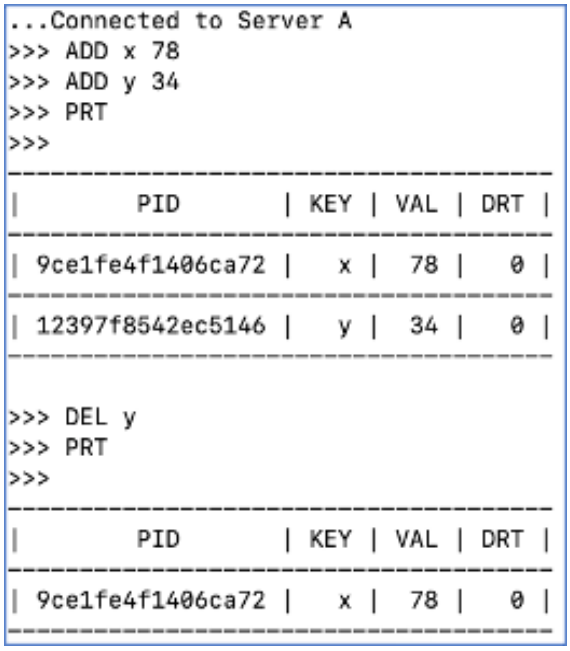

Fig. 10. Client view of database after transactions.

It was also necessary to test the ability of the system to both recognize that there exists a partition and update transactions within a partition. Therefore, by manually inducing a fault in several nodes, the system entered a partitioned state. The user subsequently adds another entry “ADD z 6”. Fig. 11 displays the result:

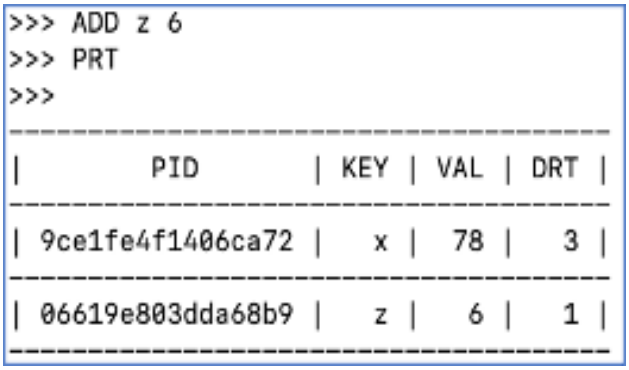

Fig. 11. Client view of database after partition.

Clearly, the entry for the key “x” has a consistency bit value of “3”, indicating to the user that this entry was once confirmed as consistent, but due to an existing partition somewhere in the system, they entry has not been changed within the partition, but cannot be confirmed to be consistent outside of the partition. Similarly, any subsequent transactions such as the addition of the key “z” have a correctly labeled consistency bit of “1”, indicating that the transaction is inconsistent globally due to the partition. The functionality of the consistency status indicator is for the verifiability of every node in the system to ensure which transactions are confirmed, consistent, and valid. Therefore, no performance experiments were conducted for this section.

6.2.3 Experimentation of Conflict Resolution Algorithm

The post-partition conflict resolution algorithm introduced in section 5.3 was tested for its performance on being able to process hundreds of thousands of transactions with a

different number of variables across the nodes. The goal of the algorithm is to evaluate every single transaction that involves each key entry in order to determine the final state of the distributed storage system after a partition in which the availability of the servers allows updates to be made while remaining inconsistent. It made sense to test the performance of this algorithm as the number of variables (unique keys) and transactions changed. After using training data to ensure that the algorithm produces the correct output, an experimentation procedure was performed by having the algorithm process a pre-given set of transactions on a single server, with results shown in Fig. 12. This is because the algorithm is independent of the number of nodes in the system since it assumes that all servers will share the entire transaction history with all other servers, thereby allowing each server to compute the algorithm individually to arrive at the same result. Therefore, this data does not include the network latency necessary to transmit the transaction history or the consensus process in the end.

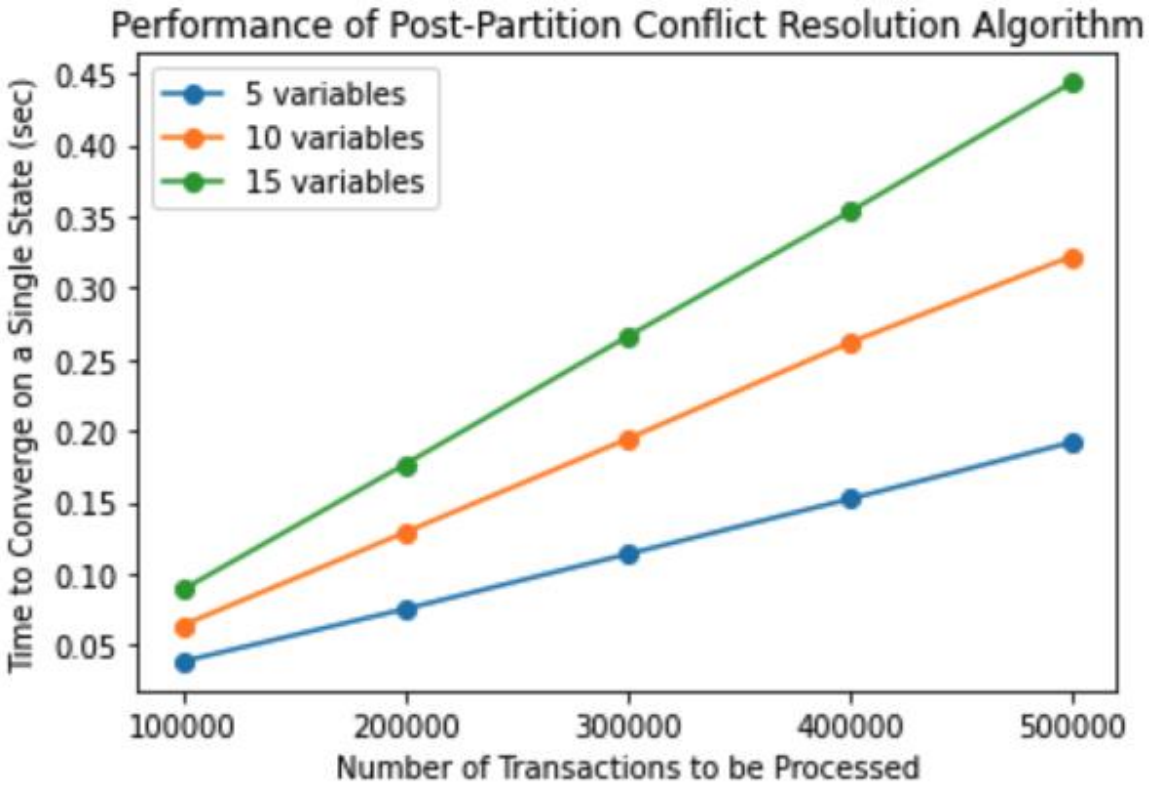

Fig. 12. Performance of conflict resolution algorithm.

For the trial with five variables, initially 100,000 random transactions were generated. The algorithm was timed on how quickly it is able to finish processing all transactions, for a total of 10 trials, all of which were averaged to produce the data point. This was then replicated for 200,000, 300,000, 400,000, and 500,000 transactions, and each for 10 and 15 variables. As can be seen from the graph, there is clearly a linear growth between the number of transactions needed to be processed and the time it takes for a server to utilize the algorithm and converge on a set of key-value pairs. However, as the number of transactions increases, the difference in convergence times also increases between different numbers of variables.

6.2.4 Experimentation of the Distributed Transaction Protocol

The distributed concurrent transaction protocol presented in section 5.4 is paramount to the propagation and atomic commitment of distributed transactions. Specifically, as it provides solutions to concurrent and conflicting transaction entries, while ensuring that the committed transaction can be verifiably consistent. The property was tested and demonstrated in the previous section, indicating that the distributed storage system using the proposed methods is able to successfully commit transactions. However, while the performance of these proposed methods relative to the performance of other implementations is irrelevant for the purposes of this thesis, it is necessary to analyze the capabilities of this system itself by testing its personal performance as the number of nodes in the system changes. Therefore, an experiment was set up such that multiple servers at random send random transactions to the rest of the servers, either conflicting or non-conflicting, with results shown in Fig. 13. By using the proposed distributed concurrent transaction protocol, any source server will send a transaction to all recipient servers in the system, wait for confirmation from all recipient servers, then after evaluating the responses, either acceptances or rejections of the transaction, it will then send a message to either abort the transaction entirely, not send anything at all, or send a message to commit the transaction across all nodes. Each server is timed to determine how long it takes for the transaction to be confirmed and replicated across the system. Each sample is repeated 10 times and the average is taken as the corresponding value for that number of nodes.

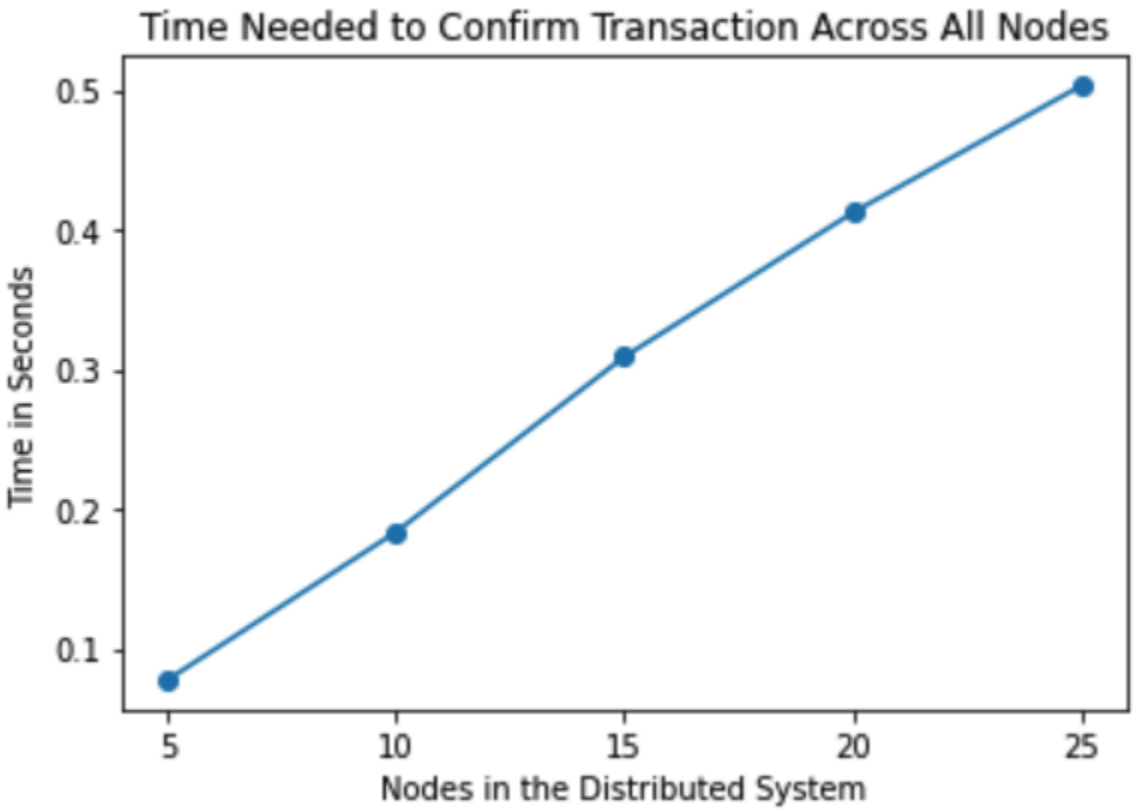

Fig. 13. Graph of transaction confirmation time.

The graph demonstrates that as the number of nodes in the system increases, the time needed to confirm a single transaction increases linearly, proportional to the size of the system. This is reasonable and expected because the original server, after receiving a signed transaction from the timestamp server, must send the transaction to all other servers in the system, wait for them to process it, then receive the transaction from all servers. Finally, after the original server processes all acceptances or rejections, it must send a confirmation message to all servers again. Therefore, it is reasonable to expect that as the number of servers increases, the confirmation time will increase.

7 DISCUSSION AND FUTURE WORK

The solutions presented in this thesis provide functional guarantees absent in previous implementations and proposals because these solutions assume various requirements regarding the system architecture, network capabilities, and database design. While there are many advantages to the proposed system architecture and implementation of distributed algorithms and processes, the drawbacks cannot be ignored, as they may not apply to other distributed application implementations. Overall, in order to improve upon consistency guarantees there will be significant tradeoffs with runtime performance.

7.1 Shortcomings of the Proposed Methods

The primary shortcoming of the proposed protocol is that it is intended for trusted, multi-endpoint, highly available distributed systems, meaning that these protocols may not provide the same verifiable consistency guarantees for other distributed systems. Secondarily, there are issues with the proposed ideas as well. The implementation of the timestamp server yields a severe bottleneck problem because every single transaction must be signed by a single server. This creates a significant computation toll on the server to process all of the transactions by mapping a timestamp and a priority ID to it. In addition to that, the timestamp server is a server itself and must process transactions from its own client. Furthermore, this will create a significant network burden on the timestamp server as the number of transactions increases. Finally, from a security perspective, if the timestamp server faults, then having to elect a new leader requires coordination which is further complicated during partitions. Therefore, this solution is primarily meant for trusted distributed systems with a few nodes, such as datacenters that transfer updates between each other. For the

consistency state indicator, during a partition, if a subgroup of servers detect that they have not formed a quorum, then because the servers do not know the status of servers that it cannot communicate with, they do not have any indication on whether other servers have formed a quorum. The conflict resolution algorithm has a fixed hierarchy of tie-breaking in the event of conflicting transactions, thereby preventing a server from committing an ADD transaction by a partitioned minority of nodes over a DEL transaction committed by a quorum. It could be improved by allowing clients to program a hierarchy of preferences for resolving conflicting transactions. Finally, the distributed coordination protocol is unable to handle coordination of a value in the event that a server faults, instead causing the entire transaction to be either canceled or to have an inconsistent state on only one of the server's databases. While this is acceptable for the purposes of verifying the consistency of that data, this event would result in an inconsistent state for a confirmed transaction. The proposals presented in this paper have demonstrated the ability to verify consistency and improve coordination and concurrency of transactions during a partition. However, for the reasons mentioned above, the ability for this protocol to work outside of the prescribed environment is largely untested and unknown regarding the scope of its performance.

7.2 Future Work

The distributed architecture considered in this thesis assumed many requirements for the implementation of the system, making it considerably different from some applications commonly used in the real world. Therefore, the obvious question remains: how would these solutions need to change when implemented on a trustless, public distributed system? A potential next step would be implementing security accommodations for an untrusted

distributed system like a decentralized blockchain [30]. One cause for concern may occur during a faulty data transmission in which a source node that disseminates the update to the recipient nodes consists of different values for the same key. This would require the involvement of public key cryptography and certificates to ensure that messages sent by one party maintain authenticity and integrity. More specifically, it becomes necessary to consider cases when the source node is malicious, when the recipient node is malicious, and when all nodes are malicious and determine measures to counteract malicious behaviors by various parties to falsify information. Additionally, this system can be improved by considering network issues in which there is no guarantee that messages certain were received. This would make the data confirmation process considerably difficult as the nodes would need to coordinate to ensure that all other nodes received and implemented every step of the update process without crashing. Adding proxy routing capabilities would mitigate some partition issues if some servers are able to communicate to a subset of nodes, a proxy, to the recipient rather than being designated as a partition. If implemented correctly, these updates algorithms could considerably improve the performance and CAP guarantees of modern distributed applications that have significant inconsistencies, adversarial threats, and storage issues.

8 CONCLUSION

This distributed storage system construction is not perfect because determining the global state of key-value pairs is extremely nuanced based on the use case, architecture, and priority policies of the system. However, this system was very practical when compared to many trusted distributed systems. Each of the solutions presented in this paper solves a major issue in contemporary distributed storage applications. The universal timestamp server allows transactions to be compared on a global level without the need for logical ordering or a mass coordination to determine relations between events. The consistency status indicator provides a relative consistency tracker for data entries that can be used by the client to verify if a transaction has achieved full consistency. The post-partition conflict resolution algorithm allows the system to remain available in the midst of a partition by allowing inconsistent updates and autonomously resolving conflicts to return the databases to a replicated state. The distributed transaction algorithm provided a low-effort coordination protocol for servers to commit conflicting transactions safely, in which transactions would either be accepted or rejected atomically. Each of the four presented proposals are interdependent and provide a useful and novel component that is necessary to achieve verifiable consistency. Ultimately, different distributed applications warrant different algorithmic implementations, and as a result, they have different performance guarantees. However, the commonalities between all distributed applications are encapsulated by the presented proposed methods. Therefore, they can be easily molded to fit the needs of any distributed application and provide guarantees for verifiable consistency.

Literature Cited

[1] C. Lepore, M. Ceria, A. Visconti, U. P. Rao, K. A. Shah, and L. Zanolini, "A Survey on Blockchain Consensus with a Performance Comparison of PoW, PoS and Pure PoS," *Mathematics*, vol. 8, no. 10, Oct. 2020, doi: 10.3390/math8101782.

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985, doi: 10.1145/3149.214121.

[3] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pp. 174–178, Mar. 1999. doi: 10.1109/HOTOS.1999.798396.

[4] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988, doi: 10.1145/42282.42283.

[5] L. Lamport and P. M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985, doi: 10.1145/2455.2457.

[6] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, Mar. 1994, doi: 10.1007/BF02277859.

[7] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, doi: 10.1145/359545.359563.

[8] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," in *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.

[9] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980, doi: 10.1145/322186.322188.

[10] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant, "A Note on Distributed Computing," Sun Microsystems, Inc., CA, USA, techreport TR-94-29, 1994.

[11] M. Kapritsos and F. P. Junqueira, “Scalable Agreement: Toward Ordering as a Service,” in *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/hotdep10/scalable-agreement-toward-ordering-service

[12] K. M. Chandy, J. Misra, and L. M. Haas, “Distributed Deadlock Detection,” *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983, doi: 10.1145/357360.357365.

[13] K. M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Trans. Comput. Syst*., vol. 3, no. 1, pp. 63–75, Feb. 1985, doi: 10.1145/214451.214456.

[14] N. M. Preguiça, C. Baquero, and M. Shapiro, “Conflict-free Replicated Data Types (CRDTs),” *CoRR*, vol. abs/1805.06358, May 2018, [Online]. Available: http://arxiv.org/abs/1805.06358

[15] M. Kleppmann and A. R. Beresford, “A Conflict-Free Replicated JSON Datatype,” *CoRR*, vol. abs/1608.03960, Aug. 2016, [Online]. Available: http://arxiv.org/abs/1608.03960

[16] B. Alpern and F. B. Schneider, “Defining Liveness,” *Inf. Process. Lett.*, vol. 21, pp. 181–185, Feb. 1985.

[17] L. Lamport, “Paxos Made Simple,” in *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001, [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[18] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pp. 305–320, Jun. 2014.

[19] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982, doi: 10.1145/357172.357176.

[20] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, “Bitcoin-NG: A Scalable Blockchain Protocol,” in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, pp. 45–59, Mar. 2016.

[21] L. Kiffer, R. Rajaraman, and abhi shelat, “A Better Method to Analyze Blockchain Consistency,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 729–744, Oct. 2018. doi: 10.1145/3243734.3243814.

[22] I. Stoica et al., “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications,” *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003, doi: 10.1109/TNET.2002.808407.

[23] R. van Renesse and F. B. Schneider, “Chain Replication for Supporting High Throughput and Availability,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation* - vol. 6, p. 7, Dec. 2004.

[24] L. Lamport and P. M. Melliar-Smith, “Byzantine Clock Synchronization,” in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 68–74, Aug. 1984. doi: 10.1145/800222.806737.

[25] J. M. Hellerstein and P. Alvaro, “Keeping CALM: When Distributed Consistency is Easy,” *Commun. ACM*, vol. 63, no. 9, pp. 72–81, Aug. 2020, doi: 10.1145/3369736.

[26] J. Gray, “The Transaction Concept: Virtues and Limitations,” *Readings in Database Systems*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 140–150, Jun. 1988.

[27] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, “A Fault-Tolerance Shim for Serverless Computing,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, Apr. 2020. doi: 10.1145/3342195.3387535.

[28] C. Wu, V. Sreekanti, and J. M. Hellerstein, “Transactional Causal Consistency for Serverless Computing,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 83–97, Jun. 2020. doi: 10.1145/3318464.3389710.

[29] R. Strom and S. Yemini, “Optimistic Recovery in Distributed Systems,” *ACM Trans. Comput. Syst.*, vol. 3, no. 3, pp. 204–226, Aug. 1985, doi: 10.1145/3959.3962.

[30] R. Zhang, R. Xue, and L. Liu, “Security and Privacy on Blockchain,” *ACM Comput. Surv.*, vol. 52, no. 3, Jul. 2019, doi: 10.1145/3316481.