Replica-centric Causal Consistency in partially replicated system

A Thesis
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Master of Science
in Computer Science

By

Dandan Wang, B.S.

Washington, DC
April 21, 2021
Replica-centric Causal Consistency in partially replicated system

Dandan Wang, B.S.

Thesis Advisor: Nitin Vaidya, Ph.D.

Abstract

Geo-replicated storage is commonly used to store data nowadays. Replication brings with it the problem of maintaining consistency across the replicas. Causal consistency is an attractive model for replicated data stores, as it allows for high availability and low latency. Maintaining causal consistency in distributed shared memory systems has received significant attention, mostly on the full replication where each replica stores a copy of all objects. Causal consistency requires that all causally dependent updates must be applied already before an operation is performed at any replica. There have been numerous mechanisms for tracking causal dependencies in full replication. For example, vector timestamp contains all replicas as the vector elements to record and check causal consistency. With the increase of data volume, full replication becomes limited deployment in practice. Hence, partial replication wherein each replica can store only a subset of all objects has become popular. However, it also leads to some problems, especially the overwhelming metadata to maintain causal consistency without false dependency involved. In this thesis, we propose a protocol which implements the replica-centric causal consistency for peer-to-peer architecture. The proposed algorithm further reduces the size of metadata and also reduces the amount of metadata exchanged in network communications.

Index words: Partially Replicated Systems, Peer-to-Peer Architecture, Causal Consistency, Distributed Systems, Key-Value Stores, YCSB
ACKNOWLEDGMENTS

I sincerely thank my advisor, Professor Nitin Vaidya who guides and encourages me during the whole process. I also wish to express my appreciation to Zhuolun Xiang who is a PhD student advised by Nitin Vaidya and provides me with useful suggestions to complete this thesis as well. I am also grateful to Professor Bala Kalyanasundaram and Professor Lewis Tseng for joining my thesis committee and their helpful advice on this thesis.
# Table of Contents

**Chapter**

1. Introduction .......................................... 1
2. Background ........................................... 4
   2.1 Consistency Models ............................... 4
   2.2 Full Replication ................................. 6
   2.3 Partial Replication ............................... 8
3. Replica-centric Consistency ............................. 11
   3.1 Preliminary Design for Peer-to-Peer Architecture Schemed by Xiang and Vaidya, 2019 ...................... 11
   3.2 Extension to Client-Server Architecture Schemed by Xiang and Vaidya, 2019 ................................ 16
   3.3 Extension to Peer-to-Peer Architecture ............. 20
   3.4 Dynamic Replication ............................... 28
4. Implementation ........................................... 32
   4.1 Distributed Key-Value Framework (DKVF) Framework ........................................ 32
   4.2 Implement Replica-centric Causal Consistency for Peer-to-Peer Architecture ....................... 33
5. Evaluation .............................................. 53
   5.1 Experimental Setup .................................. 54
   5.2 Improvement on Metadata Size ...................... 54
   5.3 Improvement on Network Communication .................. 54
   5.4 Experimental Results on Large Clusters ................ 58
6. Discussion and Conclusion .............................. 63
   6.1 Conclusion ........................................ 63
   6.2 Limitation and Future Work ....................... 63

**Bibliography** ............................................. 65
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Peer-to-peer architecture</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration for preliminary design</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustration for ((i, e_{jk}))-loop</td>
</tr>
<tr>
<td>3.3</td>
<td>Client-server architecture</td>
</tr>
<tr>
<td>3.4</td>
<td>Augmented share graph under client-server architecture</td>
</tr>
<tr>
<td>3.5</td>
<td>Augmented share graph under peer-to-peer architecture</td>
</tr>
<tr>
<td>3.6</td>
<td>An example for propagating a share graph update</td>
</tr>
<tr>
<td>3.7</td>
<td>An example for flooding data delivery</td>
</tr>
<tr>
<td>4.1</td>
<td>An example usage of Cluster Designer.</td>
</tr>
<tr>
<td>4.2</td>
<td>A share graph example</td>
</tr>
<tr>
<td>5.1</td>
<td>An example for full replication.</td>
</tr>
<tr>
<td>5.2</td>
<td>An example for partition replication.</td>
</tr>
<tr>
<td>5.3</td>
<td>A partially replicated cluster with 8 peers.</td>
</tr>
<tr>
<td>5.4</td>
<td>A partially replicated cluster with 12 peers.</td>
</tr>
<tr>
<td>5.5</td>
<td>An augmented ((2, e_{7b}))-loop for client-server architecture in Figure 5.4.</td>
</tr>
<tr>
<td>5.6</td>
<td>An example loop ((2, 3, 6, 7, 10, 5, 2)) for peer-to-peer architecture in Figure 5.4.</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Replica timestamp comparison . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.2 Exchanged edges in the fully replicated cluster . . . . . . . . . . . . . . 56
5.3 Exchanged edges in the partitioned replicated cluster . . . . . . . . . . 57
5.4 Exchanged edges in the partially replicated cluster . . . . . . . . . . . 58
5.5 Exchanged edges in the partially replicated cluster with 8 peers . . . 59
5.6 Exchanged edges in the partially replicated cluster with 12 peers . . 60
Chapter 1

Introduction

Geo-replication becomes popular in industry and academia as a design choice for large-scale data platforms to meet the strict latency and availability requirements of modern Internet services (e.g. social networking) [4]. Ideally, the distributed systems would be always available to handle requests, be able to continue providing services during network partitions, and objects are strongly consistent among all replicas. However, the CAP theorem proves that there not exists a system which can achieve all these three properties [5].

Causal consistency is an attractive consistency model for building geo-replicated data stores. On the one hand, it has intuitive semantics and avoid many abnormal scenarios allowed under weak consistency models (e.g., eventual consistency). On the other hand, it provides lower latency compared with strong consistency models (e.g., linearizability) and tolerates network partitions. Causal consistency ensures that all causally dependent updates must be already applied before an update is applied at one replica.

This thesis focuses on the peer-to-peer architecture illustrated in Figure 1.1 [1]. In peer-to-peer architecture, each peer has a client that issues read/write operations to replicas and a replica that stores the objects and values locally. That is, the client can read/write the local replica efficiently without involving network communications.

Many causally-consistent, distributed systems suffer from a key disadvantage that renders them impractical because they require full replication. In the context of full
replication, all the objects are replicated in one replica or in one data center consisting of some replicas (referred as *partitioned replication*). It becomes infeasible to implement such full replication because of the immense size of data and various storage capacities of replicas.

*Partial replication*, where each replica replicates an arbitrary subset of all objects, has been employed in recent years and also brings the question: how to track causal dependencies? One way is to add "virtual objects" at each replica to simulate full replication. However, every update message will be flooded to all replicas and unnecessary dependencies which we call *false dependency* [1] will be introduced as well. For example, if update $u_x$ on object $x$ depends on update $u_y$ on object $y$, i.e., $u_x$ can only be applied after $u_y$ is applied, then on any replica who received $u_x$ first must wait for the receipt of $u_y$, even if key $y$ is virtual and does not belong to this replica. However, there is no need to wait for $u_y$ for such replicas, because the virtual object $y$ cannot be accessed from this replica. $u_x$ can be applied directly without receiving $u_y$. The other way is to use metadata to track all dependencies. In this way, a significant amount of metadata is used to maintain causal consistency and avoid false dependencies. It is

![Figure 1.1: Peer-to-peer architecture](image-url)
challenging and interesting to find a solution to ensure causal consistency with small metadata but without false dependency in partially replicated systems.

In this thesis, we propose a protocol wherein each replica maintains an edge-indexed vector timestamp which keeps counters for a subset of edges in share graph that characterizes the partially replicated system. The protocol focuses on peer-to-peer architecture and extends the protocols in [1]. Our key contributions are:

- We propose a protocol which is more suitable for implementing causal consistency in peer-to-peer architecture.

- We present an algorithm for achieving causal consistency in peer-to-peer architecture using the metadata in above proposed protocol.

- We implement experiments on previous algorithms in [1] and our algorithm to evaluate the advantages of our proposal.
In this chapter, we introduce the popular consistency models and the related research about causal consistency in the context of full and partial replication.

2.1 Consistency Models

Geo-replication has become a popular design choice in industry and academia for large-scale data platforms to meet the strict latency and availability requirements of online applications [4]. Geo-replication ensures the high availability by placing multiple copies of data at various replicas and reduces the latency by letting clients access data in close replicas. However, such replication also brings some problems. When we replicate the same data across several replicas, we need to make sure that the client can obtain consistent data. There exist multiple levels of consistency defined for distributed systems. The CAP theorem states that it is impossible to create a system that achieves strong consistency, high availability and partition tolerance [5]. There is a trade-off between performance and consistency. Stronger consistency models require high performance overhead.

For the strong consistency, there are some models like linearizability and sequential consistency, where the replicated system behaves like a single server that serializes all operations. While there are concurrent operations in systems, the different executing order of concurrent operations does not violate the client’s consistent view of data.
Such strong consistency is not necessary for many applications. *Eventual consistency* is a weak consistency which allows replicas to diverge in a short time as long as they finally converge to the same state. Some systems achieve low latency and high availability by sacrificing the consistency, such as Dynamo [7] and Cassandra [8].

*Causal consistency* is an attractive consistency model for building geo-replicated stores. Compared with weak consistency, it preserves the intuitive happened-before relation and avoid many anomalies. Besides, it allows for high availability and avoids the long latency caused by strong consistency. Causal consistency is proved to be the strongest consistency that can be achieved in an always-available system [4, 9]. A system is called causally consistent if the return values of operations follow the *potential causality* [39, 40] which is defined by the three rules of happened-before relation for operations.

**Definition 2.1.1 (Happened-before relation → for operations [2])** Let \( o_1, o_2 \) be two operations of the client. \( o_1 \) happened-before \( o_2 \), denoted as \( o_1 \rightarrow o_2 \), if and only if at least one of the following conditions is true:

- Both \( o_1 \) and \( o_2 \) are performed by the same client, and \( o_1 \) is issued before \( o_2 \).

- \( o_1 \) is a write operation, and \( o_2 \) is a read operation that returns the value written by \( o_1 \).

- There exists operation \( o_3 \) such that \( o_1 \rightarrow o_3 \) and \( o_3 \rightarrow o_2 \).

Many works have proposed different techniques to enforce causally consistency in a geo-replicated scenario [10, 11, 12, 13]. For distributed systems, the clients participate in the communication as well and maintain metadata to ensure the causal consistency. There is an inherent trade-off between concurrency and metadata size in partially
replicated causally consistent systems. For maximum concurrency, the amount of metadata maintained by the client may be overwhelming [11]. If the metadata is compressed, false dependency may be introduced [14]. This may bring unnecessary delays and cause long latencies. Causal consistency is still the target consistency level of many systems. We introduce the techniques according to their suitability to support full replication or partial replication.

2.2 Full Replication

In the context of full replication where each geo-location replicates the full set of objects, there have been numerous designs for causally consistent distributed storage systems, including Lazy Replication [16], COPS [13], Orbe [11], ChainReaction [17], SwiftCloud [15] and GentleRain [12]. One technique to distribute objects is partitioning (also known as sharding) which allows us to store objects in more than one machine. Specifically, we could divide all objects into several parts and store each part into one machine. Usually, we partition the objects in a way that each object is assigned exactly to one partition. Partitioning increases the scalability of systems because we don’t need to store all objects in one machine.

Lazy Replication [16] is closest to COPS’s approach. It explicitly attaches updates with their causal dependencies and waits for all causal dependencies to be satisfied before the update is performed. Lazy Replication assumes that replicas are limited to a single machine: each replica requires a single point to create a sequential log of all replica operations and share the log to other replicas. Finally all replicas apply the operations in a causal order locally.

COPS [13] explicitly tracks causal dependencies of every accessed object by leveraging client support. Each update is associated with a list of dependencies. When an
update is propagated from one replica to another replica, the carried dependencies ensure the causal consistency, and this update is not performed until all updates in the dependency list have been performed locally. COPS assumes a fully-replicated system wherein all datacenters replicate the full set of data and achieves good scalability. However, explicitly tracking causal dependencies per object can cause extremely large metadata, which increases the storage and network overhead and negatively affects throughput.

Orbe [11] is also a causally consistent geo-replicated system and supports partitioned replication. Orbe maintains a vector clock with an entry per partition. Orbe aggregates the dependencies belonging to the same partition in one single scalar in the vector clock to reduce the amount of metadata to track and enforce causality.

ChainReaction [17] deals with the causal consistency within each datacenter wherein all data is stored. ChainReaction is a relaxed version of chain replication and allows concurrent causally consistent read operations. It also uses vector clocks with an entry per datacenter to achieve causally consistent geo-replication.

SwiftCloud [15] is also a causally consistent geo-replicated datastore that provides low latency reads and writes by relying on a causally consistent client-side cache. These caches are supported by a set of datastores that fully replicate all data. The implementation of causal consistency is similar to ChainReaction.

GentleRain [12], one geo-replicated causally consistent system, can achieve high throughput similar to eventually consistent systems. It proposes the global stabilization technique for maintaining causal consistency and trades off between throughput and data freshness. It slightly delays visibility by relying on a loosely synchronized physical clock. As for the metadata to track and enforce causality, GentleRain only needs a single scalar. GentleRain avoids the explicit dependency check messages, which are the major overhead of previous techniques, such as COPS.
2.3 Partial Replication

Previous approaches have successfully designed solutions under full replication. However, because of the diversity in storage capacity, computational power and network performance of datacenters, partial replication gains more attention. How to maintain causal consistency in partially replicated systems? One solution could adopt techniques designed for full-replication by adding "virtual objects" at each replica. The virtual objects are not stored at the replica actually and cannot be accessed by clients. However, this leads to some issues:

1. every update message with metadata will be sent to all replicas, which is not necessary for partial replication. This wastes the bandwidth usage.

2. False dependencies are introduced.

Fewer false dependencies also bring higher concurrency. To avoid false dependencies, more metadata are necessary to ensure the causally consistency in partially replicated systems. Partial replication yields a trade-off between concurrency and metadata size. There is some recent progress in terms of partial replication, such as PRACTI [18], Saturn [19], Karma [20], protocols proposed by Raynal and Ahamad [22], Full-Track and Opt-Track [23], and the scheme proposed by Xiang and Vaidya [1, 24].

PRACTI [18] is a mechanism to achieve causal consistency under partial replication. It allows updates to be only propagated to replicas containing the updated object. But the metadata is still sent to all servers. The metadata includes all updates to identify gaps in the causal history. Each replica has to maintain a totally ordered log of updates, which limits the scalability of the system.

Saturn [19] aims to reduce the metadata without limiting concurrency. Saturn is deployed using a set of cooperating servers, which help to distribute the client’s
load for scalability and ensure clients to access nearby replicas to achieve low latency. These servers are organized in a tree that is used to propagate causal dependencies among multiple geographic locations. All updates between data centers are serialized and transmitted through the shared tree.

In the context of partitioned replication, previous causal systems statically bind a client to its associated datacenter. Karma [20] allows a client to access any replica based on availability or latency by using novel dynamic ring binding mechanism. Karma is the first data store to support partial data replication with storage cache while offering causal consistency and availability.

Hélary and Milani [21] stated the difficulty of implementing causal consistency in partially replicated distributed systems. They proposed the notion of share graph and claimed that the size of metadata would be large for partially replicated causal consistency without false dependencies introduced. The replica-centric causal consistency protocol proposed by Xiang and Vaidya [1] improves the results of Hélary and Milani by focusing on the metadata in the replica side instead of client side, and presents tight necessary and sufficient conditions on the replica metadata. Our innovation and implementation are based on this protocol which will be introduced in details later.

Raynal and Ahamad [22] proposed protocols for partially replicated causally consistent systems under peer-to-peer architecture. The proposed algorithm uses metadata of size $O(mn)$ where $n$ is the number of replicas and $m$ is the number of objects.

Shen et al.[23] proposed two algorithms, Full-Track and Opt-Track, based on vector clocks to achieve causal consistency in the context of partial replication. The metadata size in Full-Track is $O(n^2)$. Opt-Track optimizes the Full-Track and achieves a promising amortized message size of $O(n)$ where $n$ is the number of available replicas.
Xiang and Vaidya [24] present an algorithm for partial replication which only requires metadata of constant size by using the global stabilization technique. There are also some other researches [25, 26, 27] explored how to reduce metadata (e.g., bounded timestamps) to track causal dependencies.

Overall, partial replication is attractive, but it also leads to some problems, especially the overwhelming metadata to track causal dependencies. It is an interesting and meaningful topic to reduce the size of metadata to maintain causal consistency and keep high concurrency meanwhile. In other words, it is significant to explore the necessary and sufficient metadata to track dependencies and avoid introducing false dependencies when propagating updates. As we mentioned before, false dependencies bring unnecessary latency and negatively influence concurrency.
Chapter 3

Replica-centric Consistency

Although most previous works on the implementations of causally consistent shared memory systems in the context of full replication, the research about partially replicated causally consistent shared memory systems has also attracted more attention recently. One significant topic is to achieve causal consistency without false dependencies and with less metadata. Xiang and Vaidya [1] present replica-centric causal consistency protocols with a tight necessary and sufficient condition on the metadata to implement replica-centric causal consistency in a partially replicated system. We also propose a protocol that is more suitable for peer-to-peer architecture based on the protocols in [1].

3.1 Preliminary Design for Peer-to-Peer Architecture Schemed by Xiang and Vaidya, 2019

[1] assumes an asynchronous system where the replicas communicates using reliable point-to-point message-passing channels. The preliminary protocol is designed for peer-to-peer architecture in Figure 1.1. Each peer contains a client and a replica. Assume there are $P$ peers (i.e., $P$ clients and $P$ replicas), and the IDs of replicas are 1 to $P$ (i.e., $1, 2, \ldots, P$). Each replica stores an arbitrary subset of objects. We call the set of objects in replica $i$ as $X_i$. For full replication, each replica stores the set of all objects, $X_i = X_j$ for any replica $i, j$. For partial replication, two replicas don’t
need to store totally same objects (i.e., it is possible that $X_i \neq X_j$ for replica $i, j$ and $i \neq j$) and they could share some objects as well. We call the intersection set of $X_i$ and $X_j$ as $X_{ij}$. The objects in $X_{ij}$ are stored in both replica $i$ and replica $j$. In practice, the objects belonging to one replica might change dynamically, but this protocol only takes the static scenario into consideration.

Hélary and Milani [21] introduce the notion of a *share graph* to describe the object distributions in a partially replicated shared memory system. The causal group multicast [33] also introduces similar notions of *share graphs*. This preliminary design also adopts this notion.

**Definition 3.1.1 (Share Graph [21])** *Share graph is defined as a directed graph $G = (V, E)$, where $V$ is the set of all replicas (i.e., $V = \{1, 2, ..., P\}$). We denote edge $e_{ij}$ as a directed edge from replica $i$ to replica $j$. There exist directed edges $e_{ij}$ and $e_{ji}$ in $E$ if and only if $X_{ij} \neq \emptyset$."

Since $e_{ij}$ and $e_{ji}$ are both included in the *share graph*, why not define the graph as an undirected graph? The reason is that these two edges represent different versions. $e_{ij}$ represents the dependencies propagated from replica $i$ to replica $j$, and $e_{ji}$ represents the dependencies propagated from replica $j$ to replica $i$. If a replica $k$ only tracks the dependency propagation from replica $i$ to $j$, only edge $e_{ij}$ is contained in replica $k$’s timestamp. However, if an undirected edge between $i$ and $j$ is included in replica $k$’s timestamp. It is not clear to determine that this edge tracks the dependency propagation from $i$ to $j$, or $j$ to $i$, or both. It is convenient to clarify the timestamps by using directed edges.

In this protocol, we assume that all replicas know the entire share graph. In other words, all replicas have the knowledge of object distributions among all replicas. The protocol aims to the peer-to-peer architecture, wherein each peer has a client and a
replica. The client can send read or write operations to the local replica. Figure 3.1 in [1] illustrates how the client and the replica communicate and handle operations. Define read\((x)\) to be a read operation which requests for the value of \(x\), and write\((x, v)\) to be a write operation which wants to update the value of \(x\). When client \(i\) sends read\((x)\) to the local replica \(i\), the local replica \(i\) responds the value of \(x\) stored in the local storage to client \(i\). This read operation is regarded as completed after the client receives the replica’s response. When client \(i\) sends write\((x, v)\) to the local replica \(i\), the local replica writes the \(\langle x, v \rangle\) pair into the local storage and propagates this write operation to other replicas containing \(x\) as well. The replica \(i\) also responds with an acknowledgement to client \(i\). The extra steps in the replica side are updating the timestamp and propagating this write operation to other replicas containing this key when the replica handles a write operation. The updated timestamp identifies the occurrence of the write operation. To propagate the write, we define update\((i, \tau, x, v)\), where \(i\) is the ID of the source replica, \(\tau\) is the updated timestamp, \(x\) is the object, and \(v\) is the value. Upon receiving write\((x, v)\) from the client, the replica will send updates to other replicas which also replicate \(x\) to propagate the write. When a replica receives an update, it can apply the update after the dependencies are satisfied.

![Diagram](image.png)

**Figure 3.1: Illustration for preliminary design**

Previous techniques commonly focus on the metadata in the client side. We refer this as client-centric causal consistency. Since this protocol mainly considers the metadata in the replica side, and we call it replica-centric consistency. Client-centric causal
consistency follows the relation defined in Definition 2.1.1. The updates in this protocol follows the following happened-before relation:

**Definition 3.1.2 (Happened-before relation \(\hookrightarrow\) for updates [1])** Given updates \(u_1\) and \(u_2\), \(u_1 \hookrightarrow u_2\) if and only if one of the following conditions is true:

1. \(u_1\) is applied at one replica on any object sometime before \(u_2\) is issued at the same replica on any object.

2. There exists another update \(u_3\) such that \(u_1 \hookrightarrow u_3\) and \(u_3 \hookrightarrow u_2\).

An update issued by replica \(i\) is causally dependent on all previous updates that were applied at replica \(i\), even though these updated values were not accessed by client \(i\). The paper [1] defines replica-centric causal consistency by using relation \(\hookrightarrow\).

**Definition 3.1.3 (Replica-centric Causal consistency [1])** Replica-centric causal consistency is achieved if the following two properties are satisfied:

- **Safety:** If an update \(u_1\) on object \(x\) has already been applied on replica \(i\), there must not exist an update \(u_2\) on any object \(\in X_i\). \(u_2 \hookrightarrow u_1\) and \(u_2\) has not been applied at replica \(i\) yet.

- **Liveness:** For replica \(j\), any received update \(u\) from other replicas on any object \(x \in X_j\) should be applied in a finite time after all dependencies of \(u\) have already been applied.

Research presented in [1] identifies a necessary and sufficient condition on timestamp \(\tau_i\) maintained by replica \(i\). Each replica timestamp contains a subset of directed edges in the share graph which are used to track dependencies and achieve replica-centric causal consistency. For replica \(i\), directed edge \(e_{jk}\) should be tracked if they form the following \((i, e_{jk})\)-loop (also illustrated in Figure 3.2 [1]).
Definition 3.1.4 \((i, e_{jk})\)-loop \([1]\)](i,e) Given replica \(i\) and edge \(e_{jk}(j \neq i \neq k)\) in the share graph \(G\), consider a simple loop of the form \((i, l_1, l_2, ..., l_s = k; j = r_1, r_2, ..., r_t, i)\), where \(s \geq 1\) and \(t \geq 1\). Define \(i = r_{t+1}\). This loop is an \((i, e_{jk})\)-loop if the following three conditions are all satisfied:

1. \(X_{jk} - (\cup_{1 \leq p \leq s - 1} X_{l_p}) \neq \emptyset\),
2. \(X_{jr_2} - (\cup_{1 \leq p \leq s - 1} X_{l_p}) \neq \emptyset\), and
3. for \(2 \leq q \leq t\), \(X_{r_q r_{q+1}} - (\cup_{1 \leq p \leq s} X_{l_p}) \neq \emptyset\).

![Figure 3.2: Illustration for \((i, e_{jk})\)-loop](image)

If there exists an \((i, e_{jk})\)-loop, replica \(i\) needs to track updates on edge \(e_{jk}\) to maintain causal consistency. Let \(u\) be an update issued by replica \(j\) to replica \(k\). There could be a sequence of causally dependent updates propagating along the path \((j, r_2, ..., i, ..., l_{s-1}, k)\). Let \(u'\) be an update from replica \(l_{s-1}\) to replica \(l_s\). Therefore, \(u \leftrightarrow u'\). If replica \(i\) doesn’t track the update from \(j\) to \(k\). The replica \(k\) might
apply these two updates in a order violating the causality. The details and proof are 
explained in [1].

According to the share graph and \((i, e_{jk})\)-loop, each replica could identify a set 
of directed edges to track. [1] defines the Timestamp graph \(G_i\) of replica \(i\).

**Definition 3.1.5 (Timestamp graph \(G_i\) of replica \(i\) [1])**. Given share graph 
\(G = (V, E)\), timestamp graph of replica \(i\) is defined as a directed graph \(G_i = (V_i, E_i)\), 
where \(E_i = \{e_{ij}|e_{ij} \in E\} \cup \{e_{ji}|e_{ji} \in E\} \cup \{e_{jk}|\exists (i,e_{jk})\)-loop in \(G\), \(j \neq i \neq k, e_{jk} \in E\}\), 
\(V_i = \{u, v|e_{uv} \in E_i\}\)

Replicas might have different timestamp graphs. Each replica \(i\) maintains an edge-
indexed vector timestamp \(\tau_i\) that is indexed by the edges in \(E_i\). [1] proves that this 
timestamp contains the set of necessary and sufficient directed edges to maintain 
causal consistency and presents an algorithm in peer-to-peer architecture. Maintaining 
replica-centric causal consistency doesn’t lead to large metadata because we 
use a timestamp per replica compared with some client-centric causal consistency 
protocols which keep a timestamp per object per replica. Many practical systems, 
including Lazy Replication [16], ChainReaction [10], and SwiftCloud [15], conform to 
the replica-centric consistency view.

3.2 Extension to Client-Server Architecture Schemed by Xiang and Vaidya, 2019

Above preliminary protocol for peer-to-peer architecture assumes the client \(i\) only 
accesses objects in the local replica \(i\). However, the client might access objects which 
are replicated in remote replicas. [1] also shows the extension to the client-server 
arquitectura (Figure 3.3 [1]).
Assume there are $C$ clients with IDs $\{1, 2, ..., C\}$. Client $i$ can access replicas in an arbitrary subset of all replicas (called $R_i$). Hence, client $i$ can only perform read/write operation on objects in $\bigcup_{r \in R_i} X_r$. The share graph is extended because a client may propagate dependencies across two accessible replicas. Additional edges should be added into the augmented share graph (defined below) to capture the possible dependency propagation across replicas.

**Definition 3.2.1 (Augmented Share Graph)** Augmented share graph $\hat{G} = (V, \hat{E})$, where $V = \{1, 2, ..., P\}$ and $\hat{E} = E \cup \{e_{jk} | \exists \text{ client } c \text{ such that } j, k \in R_c\}$.

In the client-server architecture, the client also needs to maintain a timestamp which will be attached with read/write operations. This client’s timestamp is used to ensure the causality of all operations issued by the client. [1] shows the modified definitions of $(i, e_{jk})$-loop and timestamp graph and also proposes the algorithm to implement replica-centric causal consistency.
Because the client can propagate causal dependencies of the updates among accessible replicas. The happened-before relation (\(\hookrightarrow\)) for replica-centric causal consistency under client-server architecture is different from the relation defined in Definition 3.1.2.

**Definition 3.2.2 (Happened-before relation \(\hookrightarrow\) for updates [1])** Given updates \(u_1\) and \(u_2\), \(u_1 \hookrightarrow u_2\) if and only if at least one of the following three conditions is true.

(i) \(u_1\) is applied at a replica before \(u_2\) is issued by the same replica.

(ii) \(u_2\) is issued by a client who previously accessed a replica that has already applied \(u_1\).

(iii) There exists another update \(u_3\) such that \(u_1 \hookrightarrow u_3\) and \(u_3 \hookrightarrow u_2\).

We will define the replica-centric causal consistency for the client-server architecture using relation \(\hookrightarrow\).

**Definition 3.2.3** Replica-centric causal consistency for client-server architecture [1] is achieved if the following two properties are satisfied:

- **Safety:** If an update \(u_1\) on object \(x \in X_i\) has already been applied on replica \(i\), there must not exist an update \(u_2\) on any object in \(X_i\) such that \(u_2 \hookrightarrow u_1\), and \(u_2\) has not been applied on replica \(i\) yet.

When replica \(i\) is accessed by a client, then there must not exist an update \(u_2\) on any object in \(X_i\) and

(i) this client previously accessed a replica that has already applied \(u_1\),

(ii) \(u_2 \hookrightarrow u_1\), and
(iii) replica i has not applied $u_2$ yet.

- **Liveness**: For replica $j$, any received update $u$ from other replicas on any object $x \in X_j$ should be applied in a finite time after all dependencies of $u$ have already been applied. Any write and read operation issued by a client to a replica will be returned in a finite time.

The causal dependencies might be propagated by the client in the client-server architecture. The previous definition of $(i, e_{jk})$-loop is not a tight condition for the client-server architecture. Therefore, [1] proposes an augmented $(i, e_{jk})$-loop for the client-server architecture.

**Definition 3.2.4 (Augmented $(i, e_{jk})$-loop [1])** Given replica $i$ and edge $e_{jk}(j \neq i \neq k)$ in the augmented share graph $\hat{G}$, consider a simple loop of the form $(i, l_1, l_2, ..., l_s = k, j = r_1, r_2, ..., r_t, i)$, where $s \geq 1$ and $t \geq 1$. Define $i = r_{t+1}$.

This loop is an augmented $(i, e_{jk})$-loop if the following three conditions are all satisfied:

(i) $X_{jk} - (\cup_{1 \leq p \leq s-1} X_{lp}) \neq \emptyset$,

(ii) $X_{jr_2} - (\cup_{1 \leq p \leq s-1} X_{lp}) \neq \emptyset$ or $j, r_2 \in R_c$ for some client $c$, and

(iii) for $2 \leq q \leq t$, $X_{rqr_{q+1}} - (\cup_{1 \leq p \leq s} X_{lp}) \neq \emptyset$ or $r_q, r_{q+1} \in R_c$ for some client $c$.

To capture the possible causal dependency propagation across the accessible replicas, additional edges are added into the augmented share graph (even though these replicas might not share any object). This augmented $(i, e_{jk})$-loop extends on condition (ii) and (iii). Recall the previous $(i, e_{jk})$-loop is to build a dependency propagation from replica $j$ to $i$ without affecting replicas $\{l_1, l_2, ..., l_{s-1}\}$. When the
client \( c \) accesses two replicas in \( r_q, 1 \leq p \leq t + 1 \) and \( r_q \in R_c \), the dependency might be propagated without affecting replicas \( l_p, 1 \leq p \leq s - 1 \). In this case, replica \( i \) needs to propagate the dependency from replica \( j \) to \( k \) as well. The definition of timestamp graph is also modified.

**Definition 3.2.5 (Augmented Timestamp Graph [1])** Given augmented share graph \( \hat{G} = (\hat{V}, \hat{E}) \), augmented timestamp graph of replica \( i \) is defined as a directed graph \( \hat{G}_i = (\hat{V}_i, \hat{E}_i) \), where \( \hat{E}_i = (\{e_{ij}|e_{ij} \in \hat{E}\} \cup \{e_{ji}|e_{ji} \in \hat{E}\} \cup \{e_{jk}|e_{jk} \in \hat{E} \text{ and } \exists \text{ augmented } (i, e_{jk}) \text{ - loop in } \hat{G}, j \neq i \neq k\}) \cap E, \text{ and } \hat{V}_i \) = \{\( u, v \)| \( e_{uv} \in E_i \}\}.

[1] gives detailed algorithm for the client-server architecture. Both the client and the replica need to maintain timestamps. Each replica \( i \) maintains a vector timestamp \( \tau_i \) indexed by edges in \( \hat{E}_i \). Each client \( c \) maintains a vector timestamp \( T_c \) indexed by edges in \( \cup_{i \in R_c} \hat{E}_i \) (i.e., all edges in the union of augmented timestamp graphs of all replicas client \( c \) can access).

### 3.3 Extension to Peer-to-Peer Architecture

The preliminary protocol has a obvious limitation that the client \( i \) can only send read/write to the local replica \( i \). Each replica only stores a subset of all objects in the partially replicated system. It is impractical that client \( i \) can only access objects in \( X_i \). The protocol for client-server architecture solves this problem. It is general to allow a client \( i \) to communicate with any replica in \( R_i \). One interesting feature of peer-to-peer architecture is that the client and the replica belong to the same peer. It is very convenient for the client to access objects from the local replica. Based on this feature and previous works, we propose a protocol which allows a client \( i \) to access any object in \( R_i \) and is more suitable for peer-to-peer architecture.
When one client can access multiple replicas in peer-to-peer architecture, we still need to consider the possible dependency propagation across replicas with the help of the client. Instead of adding edges connecting all accessible replicas in the augmented share graph (Definition 3.2.1), we could reduce the additional edges by forcing the local replica as the middle man to propagate the causal dependency. In this way, the share graph is augmented with additional edges that capture the causal dependency propagation between the local replica and the destination replicas. We add additional edges into the share graph to capture all possible dependency propagation across all replicas as well.

![Image of augmented share graph under client-server architecture](image)

**Figure 3.4: Augmented share graph under client-server architecture**

![Image of augmented share graph under peer-to-peer architecture](image)

**Figure 3.5: Augmented share graph under peer-to-peer architecture**
In Figure 3.4 and Figure 3.5, the dotted line represents the request and reply communication between the client and replicas, and the solid line represents the edges connecting two replicas in the augmented share graph. In this way, a client may propagate dependencies across replicas. As Figure 3.4 shown, the client \( c \) can access replicas \( \{i, j, k\} \). In the client-server architecture at [1], we add edges \( \{e_{ij}, e_{ji}, e_{ik}, e_{ki}, e_{jk}, e_{kj}\} \) into the augmented share graph to capture all possible dependencies propagation across replicas. However, in the peer-to-peer architecture, when client \( i \) sends a read/write request to replica \( j \) and receives reply from replica \( j \), the dependency is propagated from replica \( j \) to the local replica \( i \). Similarly, when the client \( i \) accesses replica \( k \) then, the dependency is propagated from the local replica \( i \) to replica \( k \). In this way, the dependency is also propagated across replica \( j \) and \( k \) as illustrated in Figure 3.5. Assume there are \( P \) peers and each peer contains a client and a replica. These clients and replicas are numbered 1 through \( P \). We still use previous notations. Hence, the definition of share graph under the peer-to-peer architecture should be modified into:

**Definition 3.3.1 (Augmented Share Graph for peer-to-peer architecture)**

Augmented share graph \( \tilde{G} \) consists of vertices in \( \tilde{V} = \{1, 2, ..., P\} \) and directed edges in \( \tilde{E} = E \cup \{e_{jk} | \exists k \in R_j \text{ or } j \in R_k \} \).

We call the set of all additional edges in above augmented share graph as augmented edges. We could apply the happened-before relation \( \rightarrow' \) in Definition 3.2.2 and the replica-centric causal consistency for client-server architecture in Definition 3.2.3 at the peer-to-peer architecture without modification. The definition of \((i, e_{jk})\)-loop should be modified to apply for the peer-to-peer architecture as well.

**Definition 3.3.2 (Augmented \((i, e_{jk})\)-loop for peer-to-peer architecture)**

Given replica \( i \) and edge \( e_{jk} (j \neq i \neq k) \) in the augmented share graph \( \tilde{G} \), consider a simple loop of the form \((i, l_1, l_2, ..., l_s = k, j = r_1, r_2, ..., r_t, i)\), where \( s \geq 1 \) and
\( t \geq 1 \). Define \( i = r_{t+1} \). This loop is an augmented \((i, e_{jk})\)-loop if the following three conditions are all satisfied:

(i) \( X_{jk} - (\cup_{1 \leq p \leq s-1} X_{lp}) \neq \emptyset \),

(ii) \( X_{jr_2} - (\cup_{1 \leq p \leq s-1} X_{lp}) \neq \emptyset \) or \( e_{jr_2} \in \) augmented edges, and

(iii) for \( 2 \leq q \leq t \), \( X_{r_qr_{q+1}} - (\cup_{1 \leq p \leq s} X_{lp}) \neq \emptyset \) or \( e_{r_qr_{q+1}} \in \) augmented edges.

Recall that the intuition of the \((i, e_{jk})\)-loop is to build a dependency propagation from replica \( j \) to \( i \), without affecting the state of replicas \( l_p, 1 \leq p \leq s - 1 \). The modifications on condition (ii) and (iii) identify one scenario where the dependency might be propagated by some client and not change the state of replicas \( l_p, 1 \leq p \leq s - 1 \) meanwhile. Hence the definition of timestamp graph can be naturally extended with above definition of augmented \((i, e_{jk})\)-loop for peer-to-peer architecture.

**Definition 3.3.3 (Augmented Timestamp Graph for peer-to-peer architecture)**

Given augmented share graph \( \tilde{G} = (\tilde{V}, \tilde{E}) \), augmented timestamp graph of replica \( i \) under peer-to-peer architecture is defined as a directed graph \( \tilde{G}_i = (\tilde{V}_i, \tilde{E}_i) \), where \( \tilde{E}_i = (\{e_{ij}|e_{ij} \in \tilde{E}\} \cup \{e_{ji}|e_{ji} \in \tilde{E}\} \cup \{e_{jk}|e_{jk} \in \tilde{E} \text{ and } \exists \text{ augmented } (i, e_{jk})\)-loop for peer-to-peer architecture in } \tilde{G}, j \neq i \neq k) \}\cap E, \text{ and } \tilde{V}_i = \{u, v|e_{uv} \in \tilde{E}_i\}

We also propose an algorithm based on the algorithm in [1] to implement this replica-centric causal consistency for peer-to-peer architecture where each client can access an arbitrary subset of all replicas. We use \( R_i \) to represent the set of replicas that client \( i \) can issue operations to. Each replica \( i \) maintains an edge-indexed vector timestamp \( \tau_i \) that is indexed by the edges in \( E_i \). For edge \( e_{jk} \in E_i \), \( \tau_i[e_{jk}] \) is an integer, initialized to 0. Each client \( i \) maintains a vector timestamp \( T_i \) that is indexed by the edges in the union of augmented timestamp graphs of all replicas in \( R_i \) (i.e, \( \cup_{j \in R_i} \tilde{E}_j \)).
In this thesis, we use $\tau$ to represent the replica’s timestamp, and use $T$ to represent the client’s timestamp.

Compared with the client-server architecture, the augmented share graph for peer-to-peer architecture may have fewer additional edges and the replica in peer-to-peer architecture may track fewer edges to maintain the causal consistency as well. Intuitively, since we force the local replica as the middle man to propagate the causal dependencies, the paths to propagate dependencies become fewer. Hence, the replica can ensure the replica-centric causal consistency for peer-to-peer architecture by tracking fewer edges. In order to reduce the amount of exchanged metadata, we give the local replica a higher priority. If the key is replicated at the local replica, the client should access the local replica rather than a remote one. As the algorithm for client-server architecture in [1] presents, the attached timestamp with the client’s request is the entire client timestamp which is unnecessary. In the algorithm for peer-to-peer architecture, we only exchange edges in the destination replica’s timestamp which are adequate to maintain causality.

**Client $i$’s algorithm (for peer-to-peer architecture):**

Client $i$ maintains a timestamp $T_i$ and sends read/write requests on any object $x \in \cup_{j \in R_i} X_j$.

- When client $i$ wants to read an object $x$, client $i$ checks whether this object is replicated at the local replica $i$ ($x \in X_i$) or not. If yes, choose the local replica as the destination replica $k$ (i.e., $k = i$). Otherwise, randomly choose a remote replica in $R_i$ containing $x$ as the destination replica $k$. Client $i$ sends $read(x, i, T)$ request to replica $k$ and awaits replica $k$’s response containing $x$’s value and a timestamp $\tau_k$. (The attached timestamp $T$ is a subset of $T$ and only contain all edges in $\tau_k$.) Then client $i$ updates its timestamp using $merge_1$.
If the destination replica $k$ is not the local replica, client $i$ needs to issue an update $update(i, T)$ to the local replica. (Here, $T$ only contains edges in $\tau_i$.) The client finishes the operation only after the update returns.

- When client $i$ wants to send a write $(x, v)$ request, client $i$ checks whether this object is replicated at the local replica $i$ ($x \in X_i$) or not. If yes, choose the local replica as the destination replica $k$ (i.e., $k = i$). Otherwise, randomly choose a remote replica from $R_i$ containing $x$ as the destination replica $k$. Client $i$ sends the $write(x, v, i, T)$ request to replica $k$ and awaits replica $k$'s response containing a timestamp $\tau_k$. Then client $i$ updates its timestamp using $merge_1$ function as $T_i = merge_1(\tau_k, T_i)$. If the destination replica $k$ is not the local replica, client $i$ needs to issue an update $update(i, T)$ to the local replica as well. Similarly, the client completes the operation only after the update returns.

**Replica $i$’s algorithm (for peer-to-peer architecture):**

Replica $i$ maintains a timestamp $\tau_i$.

1. When replica $i$ receives a $read(x, k, T)$ request from client $k$, the request is buffered until predicate $J_1(i, \tau_i, k, T)$ presented later evaluates true. Once the predicate evaluates true, replica $i$ responds to client $k$ with the value of object $x$ in the local storage and its timestamp $\tau_i$, and removes $read(x, k, T)$ from the buffer.

2. When replica $i$ receives a $write(x, v, k, T)$ request from client $k$, the request is buffered until predicate $J_1(i, \tau_i, k, T)$ evaluates true. Once the predicate evaluates true, replica $i$ performs the following steps atomically:
(i) write $v$ into the local storage, and update its timestamp $\tau_i$ using function $advance$ that is presented later, as $\tau_i = advance(i, \tau_i, k, T)$.

(ii) send $update(i, \tau_i, x, v)$ to all other replicas $j$ containing $x$ ($x \in X_j$), and

(iii) return timestamp $\tau_i$ to client $k$.

and removes $write(x, v, k, T)$ from the buffer.

3. When replica $i$ receives a message $update(i, T)$ from the local client $i$, the update is buffered until predicate $J_1(i, \tau_i, i, T)$ evaluates true. Once the predicate evaluates true, replica $i$ updates its timestamp $\tau_i$ using $merge_2$ function presented below as $\tau_i = merge_2(\tau_i, T)$ and removes $update(i, T)$ from the buffer.

4. When replica $i$ receives a message $update(k, \tau_k, x, v)$ issued by replica $k$, the update is buffered until predicate $J_2(i, \tau_i, k, \tau_k)$ presented below evaluates true. Once the predicate evaluates true, replica $i$ writes value $v$ to the local storage, updates its timestamp $\tau_i$ using $merge_3$ function presented below as $\tau_i = merge_3(i, \tau_i, k, \tau_k)$, and removes $update(k, \tau_k, x, v)$ from the buffer.

Now we specify the predicates $J_1$, $J_2$, functions $advance$ and $merge_1$, $merge_2$, $merge_3$ mentioned above.

- Predicate $J_1(i, \tau, k, T) = true$ if and only if $\tau[e_{ji}] \geq T[e_{ji}]$, for each $e_{ji} \in \tilde{E}_i$.

- Predicate $J_2(i, \tau_i, k, \tau_k) = true$ if and only if $\tau_i[e_{ki}] = \tau_k[e_{ki}] - 1$ and $\tau_i[e_{ji}] \geq \tau_k[e_{ji}]$, for each $e_{ji} \in \tilde{E}_i \cap \tilde{E}_k, j \neq k$.

- Function $advance(i, \tau, k, T)$ at replica $i$ returns vector $\tau$ (indexed by edges in $\tilde{E}_i$) defined as follows. For each $e_{ji} \in \tilde{E}_i$: 26
\( \tau_{ejl} := \begin{cases} 
\tau_{ejl} + 1, & \text{if } j = i \text{ and } x \in X_{il}, \\
\max(\tau_{ejl}, T[e_{jl}]), & \text{otherwise.} 
\end{cases} \)

- Function \( merge_1(\tau, T) \) at client \( i \), returns following vector \( T \) (indexed by edges in \( \bigcup_{j \in \mathcal{R}_i} \bar{E}_j \)):

\[
T[e] := \begin{cases} 
\max(T[e], \tau[e]), & \text{for each edge } e \in \bar{E}_i, \\
T[e], & \text{for each edge } e \in (\bigcup_{j \in \mathcal{R}_i} \bar{E}_j) - \bar{E}_i.
\end{cases}
\]

- Function \( merge_2(\tau, T) \) at replica \( i \), returns following vector \( \tau \) (indexed by edges in \( \bar{E}_i \)):

\[
\tau[e] := \max(T[e], \tau[e]), \quad \text{for each edge } e \in \bar{E}_i
\]

- Function \( merge_3(i, \tau_i, k, \tau_k) \) at replica \( i \) returns vector \( \tau \) (indexed by edges in \( \bar{E}_i \)):

\[
\tau[e] := \begin{cases} 
\max(\tau_i[e], \tau_k[e]), & \text{for each edge } e \in \bar{E}_i \cap \bar{E}_k, \\
\tau_i[e], & \text{for each edge } e \in \bar{E}_i - \bar{E}_k.
\end{cases}
\]

For the preliminary design presented above, [1] shows the detailed proof for the sufficiency of tracking edges in timestamp graph (Definition 3.1.5). The procedures to issue an update by one replica to another replica and handle the received update from one replica are identical in the preliminary design (Section 3.1) and this extension to peer-to-peer architecture.

Different from the algorithm for client-server architecture, the algorithm for peer-to-peer architecture adds fewer additional edges in the augmented share graph to propagate dependencies. However, it can achieve replica-centric causal consistency as well. Take an example where client \( i \) accesses replica \( j \) and \( k \). Edge \( e_{jk} \) is added into the
augmented share graph for client-server architecture. The dependency can be propagated via $e_{jk}$ directly. Edges $e_{ji}$ and $e_{ik}$ are added into the augmented share graph for peer-to-peer architecture. When client $i$ receives replica $j$’s timestamp because of issued GET/PUT operation to replica $j$, the dependencies are updated to the local replica $i$. Then the dependencies can be propagated to replica $k$ via $e_{ik}$ as well. Intuitively, peer $i$ is the middle man to propagate the dependency, hence replacing the edge $e_{jk}$ with $e_{ji}$ and $e_{ik}$.

The *edge-indexed* timestamp ensures the update message from/to replica $i$ are performed sequentially and also implies that replica $i$ is not oblivious to update on any edge $e_{jk} \in \tilde{E}_i$. In the next chapter, we describe an implementation of the protocol presented above.

3.4 Dynamic Replication

The keys replicated in a replica can be changed dynamically. Above protocols have not taken such dynamic scenarios into consideration, so we discuss how to allow dynamic replication in partially replicated systems. Assume that there is reliable FIFO delivery among replicas. When new keys are added into a replica or some keys are removed from a replica (called this a *share graph update*), we need to flood and propagate the update to all other replicas. The flooding for data delivery is a notion of routing protocols. The sender replica $i$ broadcasts the update to all its neighbors.

- If new keys are added into replica $i$, the share graph update should be sent to all neighbors in the updated share graph.

- If some keys are removed from replica $i$, the share graph update should be sent to all neighbors in the old share graph. Otherwise, some replicas may be unable to receive the share graph update. Assume that replica $i$ shares keys with replica
and $k$ originally (illustrated in Figure 3.6). Then replica $i$ removes some keys and doesn’t share keys with replica $j$. If replica $i$ only sends the corresponding share graph update to neighbors in the updated share graph (i.e., replica $k$), then the original neighbor replica $j$ will never receive this share graph update. Hence, the share graph update is sent to all neighbors in the old share graph when some keys are removed from replica $i$.

Each replica $j$ receiving the share graph update applies it (i.e., updating the share graph and its timestamp graph) and then forwards it to all neighbors in the updated share graph. These neighbors share keys with replica $j$ which already applied the share graph update. Besides, a sequence number (i.e., the source replica ID and an increasing integer) is used to avoid one replica forwarding the same share graph update multiple times. When one replica receives a share graph update, it checks whether this update has already been applied or not according to the attached sequence number. If applied, ignore this received one; if not, apply and forward it. When a replica applies a share graph update, it updates the share graph and its timestamp graph as well. The updated timestamp will be used to maintain causal consistency in the future. When all replicas don’t forward the share graph update, the flooding for the share graph update is completed.

Can the flooding for data delivery and FIFO delivery maintain the replica-centric causal consistency? The answer is yes. As illustrated in above proposed algorithms, the update issued by one replica can only be sent to other replicas containing the key as well. The share graph update is sent through all possible propagating paths. Suppose that an update $u$ on the new added keys is issued after the share graph update by the sender replica $i$ and sent to some neighbors which also replicate this key locally. Flooding for data delivery makes sure that the share graph update is broadcast to other replicas via all possible paths. $u$ is propagated via some paths as
Figure 3.6: An example for propagating a share graph update

well, and FIFO delivery guarantees that the share graph update must arrive before $u$. Hence, we receive and apply the share graph update firstly (i.e., update the share graph and the timestamp graph). Then $u$ arrives and is applied if and only if all causal dependencies are satisfied. When an update is applied at one replica, the timestamp must contain the necessary and sufficient edges in the corresponding timestamp graph. In other words, when an update is applied at one replica, the previous share graph updates must be already applied at this replica.

Take Figure 3.7 as the example to illustrate how to support dynamic replication. Suppose that replica 1 ($r_1$) changes the keys replicated locally. The share graph $G$ is also changed. This modification of the share graph (called $u'$) is sent to all its neighbors - $r_2, r_3, r_4, r_7$. If these neighbors receive $u'$ at the first time, $u'$ is applied and forwarded by them to their neighbors as well. $r_4$ may receive the forwarded $u'$ from $r_2$ after receiving $u'$ from $r_1$. $r_4$ ignores the $u'$ from $r_2$ instead of forwarding it to neighbors. Finally, all replicas receive this update $u'$. An update $u$ is issued by $r_1$ after $u'$ and propagated to $r_4$. No matter which path is chosen to propagate $u$, for
example, $r_1 \rightarrow r_4$, $r_1 \rightarrow r_2 \rightarrow r_4$ or $r_1 \rightarrow r_2 \rightarrow r_5 \rightarrow r_4$, $u'$ must be already applied when $u$ is handled. This is a property of FIFO delivery. $u'$ is broadcast via all possible paths and arrives at each accessible replica as soon as possible. In this way, we can change the keys replicated at each replica easily and update the timestamp graph of each replica without violating the replica-centric causal consistency as well.
Chapter 4

Implementation

4.1 Distributed Key-Value Framework (DKVF) Framework

We implement all protocols on the Distributed Key-Value Framework (DKVF) [28] which is proposed by Roohitavaf and Kulkarni and enables protocol developers to quickly create prototypes running their protocols to see how they work in practice. DKVF adopts key-value stores which provide a simple abstraction to store and retrieve data as the storage engine. Each key-value store is a set of \( \langle \text{key}, \text{value} \rangle \) pairs. It supports two basic operations: PUT\((k,v)\) and GET\((k)\). PUT\((k,v)\) writes the value \(v\) of key \(k\) into the storage, and GET\((k)\) reads the value of key \(k\) from the storage. Key-value stores support multi-version values or single-version value. In the multi-version type, a version chain is maintained for each key. While in the single version, the previous values are overwritten by the new value. Key-value stores are the basis of document-oriented databases which can efficiently handle one-to-many relations compared with relational databases. There are many storage systems supporting documented-oriented data model, such as MongoDB [29], CouchDB [30], and Espresso [31]. DKVF can easily change the storage engine of the low-level key value store without changing the high-level protocol.

Besides, DKVF uses Yahoo! Cloud Serving Benchmark (YCSB) [32] to generate the workloads according to defined properties to evaluate different storage systems. This provides a standardized method to evaluate protocols. There are also some other
tools to establish this framework, for instance, Cluster Designer [28] and Cluster Manager [28]. Cluster Designer, like Figure 4.1 is a graphical tool which allows us to define the cluster and experiments easily. Cluster Manager is a command line application which supports us to load the cluster and run experiments with commands easily. It also reports the cluster status and experimental results. Besides, some protocols (such as COPS, CausalSpartan [41], GentleRain, Orion [42] and Eventual consistency) have already been implemented in DKVF. It is convenient to compare these protocols with our protocols by using DKVF in the future.

Figure 4.1: An example usage of Cluster Designer.

More details about high-level interfaces provided by DKVF and DKVF’s low-level implementations will be introduced in Section 4.2.

4.2 Implement Replica-centric Causal Consistency for Peer-to-Peer Architecture

In this section, we will introduce how to implement the replica-centric causal consistency for peer-to-peer architecture using DKVF. The proposed algorithm identifies timestamp with the knowledge of key distributions of all replicas (i.e., augmented
share graph $\tilde{G}$). Since the amount of keys in the workload is large, it is impractical to point out every key replicated at one replica. We use hash mapping to classify all keys into several buckets. We only need to clarify the buckets each replica replicates in the configuration file. In DKVF, we use MD5 to calculate the key’s hash and then map the key into the corresponding bucket via modulus. In addition to MD5, we can use any hash function as long as the function is consistent in both client and replica side.

To develop the protocol in DKVF, we need to specify three components: metadata description, the client side of the protocol, and the server side of the protocol.

### 4.2.1 Metadata Description

DKVF relies on Google Protocol Buffers [35] (also called protobuf) for marshalling/unmarshalling data for storage and transmission. Hence, we can describe metadata by writing a `.proto` text file which contains a set of message blocks. Each message has a set of fields and each field has a type that is either a primitive type (e.g., integer), or another message. For protocols in DKVF, there must be four messages in the metadata: `Record`, `ClientMessage`, `ServerMessage`, and `ClientReply`. 
message Edge {
  int32 vertex1 = 1;
  int32 vertex2 = 2;
}

message Dependency {
  Edge edge = 1;
  int64 version = 2;
}

message Record {
  bytes value = 1;
}

message PutMessage {
  string key = 1;
  Record value = 2;
  repeated Dependency timestamps = 3;
}

message ClientMessage {
  oneof message_type {
    GetMessage get_message = 1;
    PutMessage put_message = 2;
    TimestampMessage t_message = 3;
    UpdateTMessage update_t_message = 4;
  }
}

Listing 4.1: Example messages in metadata.proto [36]
Record describes the format of stored (key, value); ClientMessage describes requests issued by clients; ServerMessage describes server messages among servers; and ClientReply describes the response issued by a replica to a client. Consider the protocol we proposed in Section 3.3. There is no need to maintain multi-version values of one key. There are four kinds of client messages: Get Message, Put Message, Timestamp Message, and Update Timestamp Message. The client \( i \) requests for the timestamps of replicas in \( R_i \) via Timestamp Message. Client \( i \) issues update to the local replica \( i \) via Update Timestamp Message. There are also four types of corresponding client replies: Get Reply, Put Reply, Timestamp Reply and Update Timestamp Reply. For the server message, there is only one type - Replicate Message which represents the \( update(i, \tau_i, x, v) \) mentioned in above algorithm.

The above listing shows some example messages in the metadata description for replica-centric causal consistency for peer-to-peer architecture. The entire protobuf description is provided in [36].

4.2.2 Client Side Implementation

To implement the client side, we use a class ReplicaCentricClient to extend the abstract class DKVFClient. There are some global variables in ReplicaCentricClient class:

- \( \text{clientId} \) is the ID of this client;

- \( \text{replicas} \) is an array to store IDs of all replicas this client can access (i.e., client \( i \)'s \( \text{replicas} = R_i \));

- \( \text{timestamp} \) is a hash map containing all edges in the union augmented timestamp graph of replicas in \( \text{replicas} \), and each edge’s version;
• *replicaKeys* is an array of hash sets, and each hash set contains the buckets replicated in a replica.

Hash map is a data structure which maps keys to values and provides efficient lookup. We use hash map to get the corresponding version of a tracking edge quickly. Hash set is a data structure which stores unique elements and also provides efficient lookup. We use hash set to store the buckets replicated at one replica and hash set makes it easier to check whether this replica contains the key in client’s request or not. The replicas in *replicas* correspond to the replicas in *replicaKeys*. If *replicas* = \{1, 3, 5\}, the first, second and third hash set in *replicaKeys* contain keys in replica 1, 3 and 5 respectively. There are two main interfaces: \textit{put} (Algorithm 1) and \textit{get} (Algorithm 2) for the client’s write and read requests. These two interfaces rely on some other functions, e.g., \textit{localReplicaContains} which returns true if the key is replicated at the local replica, \textit{randomReplica}, \textit{merge} and \textit{updateLocalReplica}. The \textit{Get Message} contains the \textit{key} and also attaches the client’s \textit{timestamp}; the \textit{Put Message} contains \textit{⟨key, value⟩} and also attaches the client’s \textit{timestamp}. As we mentioned before, the attached timestamp in requests is not the client’s entire timestamp and only contains the edges in the destination replica’s timestamp. The code for client side implementation is available in [36].
In the construction function of ReplicaCentricClient class, we assign values to global variables clientId and replicas. Before issuing the first operation, the client needs to request for each accessible replica’s timestamp and replicated buckets to construct its timestamp and replicaKeys (Algorithm 3).
Algorithm 3 requestTimestamps()

1: construct a timestamp message \( tm \) which requests for the replica’s timestamp;
2: \textbf{for} serverId: \( \text{replicas} \) \textbf{do}
3: \hspace{1em} sendToServer(serverId, \( tm \));
4: \hspace{1em} ClientReply cr = readFromServer(serverId);
5: \hspace{1em} \textbf{for} Dependency dep: cr.timestampList \textbf{do}
6: \hspace{2em} \textbf{if} !timestamp.contains(dep.Edge) \textbf{or} timestamp[dep.Edge]<dep.Version \textbf{then}
7: \hspace{3em} timestamp.add([dep.Edge], dep.Version);
8: \hspace{2em} \textbf{end if}
9: \hspace{1em} \textbf{end for}
10: \hspace{1em} replicaKeys.add(cr.keyList);
11: \textbf{end for}

If the \textit{key} of put/get request is not replicated in the local replica \textit{clientId}, we choose a random destination replica in \textit{replicas} containing \textit{key} using Algorithm 4 as follows. Suppose the IDs array \textit{replicas} is viewed as a loop. Then choose a random index of \textit{replicas} and then return the closest following replica containing \textit{key}.

Algorithm 4 randomReplica (key)

1: generate a random integer \( \text{randId} \) in range(0, \text{replicas.size()-1});
2: \textbf{for} \( i \leftarrow \text{randId}; \ i<\text{randId} + \text{replicas.size()}; \ i++ \) \textbf{do}
3: \hspace{1em} \textbf{if} \text{replicas}[\text{mod} \text{replicas.size()}] contains \text{key} \textbf{then}
4: \hspace{2em} \textbf{return} \text{replicas}[(\text{mod} \text{replicas.size()})];
5: \hspace{1em} \textbf{end if}
6: \textbf{end for}

After receiving the destination replica’s timestamp, the client updates its timestamp using \textit{merge} (Algorithm 5).

Algorithm 5 merge(timestampList)

1: \textbf{for} Dependency dep: timestampList \textbf{do}
2: \hspace{1em} \textbf{if} dep.Version>timestamp[dep.Edge] \textbf{then}
3: \hspace{2em} timestamp[dep.Edge] \leftarrow \text{dep.Version};
4: \hspace{1em} \textbf{end if}
5: \textbf{end for}
When the client accesses a remote replica and receives the replica’s timestamp, the client calls `updateLocalReplica` (Algorithm 6) to issue the update to the local replica and awaits the acknowledgement.

**Algorithm 6** updateLocalReplica(timestampList)

1: construct a update timestamp message `utm`;
2: ```sendToServer(clientId, utm)```;
3: ```ClientReply cr ← readFromServer(clientId)```;

4.2.3 Server Side Implementation

To implement the server side, we use a class `ReplicaCentricServer` to extend the abstract class `DKVFServer`. There are some global variables in `ReplicaCentricServer` class:

- `serverId` is the ID of this replica;
- `numOfServers` is the number of replicas in the cluster;
- `timestamp` is a hash map containing the edges to be tracked and their versions;
- `AdMatrix` is the adjacent matrix to represent the shared buckets among all replicas;
- `pendingReplicateMessages` is a list to store all pending replicate messages (i.e., updates issued by another replica) which cannot be handled right now;
- `pendingClientMessages` is a similar list to store all pending client messages which cannot be handled right now;
- `shareGraph` is a list of hash sets and each set stores a set of buckets located in one replica;
• `augmentedEdges` is a hash set to store additional edges in the augmented share graph for peer-to-peer architecture.

Take an example (Figure 4.2) to illustrate the `shareGraph` and `AdMatrix`. In this share graph, there are four replicas (1, 2, 3, 4) and four clients (1, 2, 3, 4). Replica 1 contains bucket [0, 1] and client 1 can access replica 1, 3, and 4; replica 2 contains bucket [1, 2] and client 2 can only access replica 2; replica 3 contains bucket [2] and client 3 can access replica 3; replica 4 contains bucket [0, 3] and client 4 can access replica 1 and 4. `shareGraph` is equal to [[0, 1], [1, 2], [2], [0, 3]] and `AdMatrix` is equal to [[0, [1], [0], [2], [0], [0], [2], [0], [0], [0]], [0], [0], [0]]. The elements of `AdMatrix`’s diagonal are empty sets. That is, the adjacent set between the same replica is empty (e.g., `AdMatrix[1][1] = ∅`). Other elements in `AdMatrix` represent the set of shared buckets between two replicas (e.g., `AdMatrix[0][1] = [1]` and `AdMatrix[0][2] = ∅`).

![Figure 4.2: A share graph example](image)

The code for server side implementation is available in [37]. Each replica knows the key distributions of all replicas in the cluster. In the construction function of `ReplicaCentricServer` class, we identify and add all necessary and sufficient edges into `timestamp` according to the Definition 3.3.3 (as shown in Algorithm 7, 8, 9, 10, 11). In Algorithm 7, we assign values to global variables by reading from the configuration files, and call `generateDependencies` (Algorithm 8) where we construct
the adjacent matrix and extract augmented edges. Then we use Depth First Search (DFS) algorithm (Algorithm 9 and Algorithm 10) to detect all loops containing the replica serverId. We then use checkEdges (Algorithm 11) to identify the edges to track according to the definition of the augmented \((i, e_{jk})\)-loop for peer-to-peer architecture and add these edges into timestamp. The edge set of augmented timestamp graph for peer-to-peer architecture (Definition 3.3.3) only contains directed edges that also belong to the share graph \(G\). Edges in the augmented timestamp graph for peer-to-peer architecture but not in the share graph are not contained in timestamp. Because the update can be only propagated between replicas via shared keys which are represented by directed edges in \(E\). The step 9 of Algorithm 7 achieves this by removing unnecessary edges.

**Algorithm 7** ReplicaCentricServer

1: assign values to global variables - serverId, numOfServers, and shareGraph from configuration files;
2: for \(i \leftarrow 1; i \leq \text{numOfServers}; i++\) do
3:     for \(k: R_i\) do
4:         augmentedEdges \leftarrow Edge\((i, k)\);
5:         augmentedEdges \leftarrow Edge\((k, i)\);
6:     end for
7: end for
8: generateDependencies();
9: remove edges not in the edge set \(E\) of share graph \(G\);

Steps 12-22 in Algorithm 8 add the edges between two replicas with shared buckets or same client into timestamp. This identifies the first two sets in the edge set of augmented timestamp graph for peer-to-peer architecture.
Algorithm 8 generateDependencies()
1: for $i \leftarrow 0; i < numOfServers; i ++$ do
2: \hspace{1em} AdMatrix[i][i] \leftarrow \emptyset;
3: \hspace{1em} for $j \leftarrow i + 1; j < numOfServers; j++$ do
4: \hspace{2em} AdMatrix[i][j] \leftarrow$ the intersection edge set of replica $i+1$ and $j+1$;
5: \hspace{1em} end for
6: end for
7: for $i \leftarrow 0; i < numOfServers; i ++$ do
8: \hspace{1em} for $j \leftarrow 0; j < i; j++$ do
9: \hspace{2em} AdMatrix[i][j] \leftarrow AdMatrix[j][i];
10: \hspace{1em} end for
11: end for
12: for $i \leftarrow 0; i < numOfServers; i ++$ do
13: \hspace{1em} if AdMatrix[serverId – 1][i] not empty then
14: \hspace{2em} timestamp.add(Edge(serverId, i + 1), 0);
15: \hspace{2em} timestamp.add(Edge(i + 1, serverId), 0);
16: \hspace{1em} end if
17: end for
18: for Edge $e$: augmentedEdges do
19: \hspace{1em} if $e$.Vertex1 is serverId or $e$.Vertex2 is serverId then
20: \hspace{2em} timestamp.add($e$, 0);
21: \hspace{1em} end if
22: end for
23: loop();

Algorithm 9 loop()
1: path.add(serverId);
2: Initialize a boolean array - visited with numOfServers False.
3: visited[serverId-1] \leftarrow$ true; // For replica $i \in [1, numOfServers]$, the corresponding index in visited array is $i – 1 \in [0, numOfServers – 1]$.
4: DFSHelper(path, visited, serverId);

\textit{checkEdges(path)} illustrates how we determine whether the edge $e_{\text{path}[k+1]\text{path}[k]}$ should be tracked or not according to the definition of the augmented $(i, e_{jk})$-loop for peer-to-peer architecture. We use a helper function $\text{distinct}(a, b)$ to evaluate whether set $a$ contains distinct keys from $b$ or not. The step 4 in Algorithm 11 corresponds the first and second condition in the augmented $(i, e_{jk})$-loop for peer-to-peer architecture.
Algorithm 10 DFSHelper(path, visited, Id)

1: for i ← 1; i ≤ numOfServers; i ++ do
2:  if AdMatrix[i-1][Id-1] not empty or augmentedEdges.contains(Edge ← (Id, i)) then
3:    if i = serverId then
4:      if path.size() ≥ 3 then
5:        path.add(serverId);
6:        checkEdges(path);
7:        path removes the last element-serverId;
8:    else if not visited[i-1] then
9:      path.add(i);
10:     visited[i-1] ← true;
11:    DFSHelper(path, visited, i);
12:   path removes the last element-i;
13:   visited[i-1] ← false;
14:  end if
15: end if
16: end if
17: end for

The set union contains the union of all keys in X_{path[p]} where 1 ≤ p ≤ k − 1. If these conditions are satisfied, we need to extend the set union by adding the keys in X_{path[k]} (shown in step 5). The for loop in step 6-15 corresponds to the third condition in the augmented (i, e_{jk})-loop for peer-to-peer architecture. It checks whether replica path[j] and path[j + 1] (where k + 2 ≤ j ≤ path.size() − 2) share distinct keys from union (i.e., ∪_{1≤p≤k} X_{path[p]}) or whether there exists a client to propagate dependency across replica path[j] and path[j + 1]. The edge is added into timestamp if all these three conditions are satisfied.

Since DKVF follows an event-driven approach to define a protocol [28]. We define the protocol as a set of event handlers. We need to override two main event handlers: handleClientMessage and handleServerMessage of DKVFServer class. In han-
Algorithm 11 checkEdges(path)
1: $k \leftarrow 1$;
2: union $\leftarrow$ empty hash set;
3: while $k + 2 < \text{path.size()}$ do
4: \[\text{if } \neg \text{timestamp contains Edge}(\text{path}[k+1], \text{path}[k]) \text{ and}
\quad \text{distinct(AdMatrix[ path[k]-1][path[k+1]-1], union)} \text{ and}
\quad \text{distinct(AdMatrix[ path[k+1]-1][path[k+2]-1], union)} \text{ or}
\quad \text{augmentedEdges.contains(Edge\{ path[k+1], path[k+2]\})} \text{ then}
5: \quad \text{union.add(shareGraph[ path[k]-1]);}
6: \quad j \leftarrow k + 2;
7: \quad \text{for } ; j \leq \text{path.size()-2}; j + + \text{ do}
8: \quad \quad \text{if } \neg \text{distinct(AdMatrix[ path[j]-1][path[j+1]-1], union)} \text{ and } \neg \text{augmentedEdges.contains(Edge\{ path[j], path[j+1]\})} \text{ then}
9: \quad \quad \quad \text{for } l \leftarrow k + 1; l \leq j; l + + \text{ do}
10: \quad \quad \quad \quad \text{union.add(shareGraph[ path[l]-1])}
11: \quad \quad \text{end for}
12: \quad k \leftarrow j;
13: \quad \text{break;}
14: \text{end if}
15: \text{end for}
16: \text{if } j = \text{path.size()-1} \text{ then}
17: \quad \text{timestamp.add(Edge\{ path[k+1], path[k]\}, 0);}
18: \text{end if}
19: \text{else}
20: \quad \text{union.add(shareGraph[ path[k]-1]);}
21: \text{end if}
22: \quad k \leftarrow k + 1;
23: \text{end while}
dleClientMessage, we have four event handlers for different types of client messages (Algorithm 12); in handleServerMessage, we also have the event handler for replicate message (Algorithm 20).

**Algorithm 12** handleClientMessage(ClientMessageAgent cma)

1: if get message then
2:   handleMessage(cma);
3: else if put message then
4:   handlePutMessage(cma);
5: else if timestamp message then
6:   handleTimestampMessage(cma);
7: else if update timestamp message then
8:   handleUpdateTimestampMessage(cma);
9: end if

When the replica receives a get request from the client, it checks whether the message can be responded right now or not. If all causal dependencies of this request have been applied at the replica, the replica can read from the local storage and reply the value to the client. If some causal dependencies have not been applied yet, add this client message into pendingClientMessages and wait for handling it later (Algorithm 13 and Algorithm 14). Algorithm 19 achieves $J_1$ mentioned in the proposed algorithm in Section 3.3 to check whether the request should be handled right now or not. Similarly, when the replica receives a put request from the client, it also needs to check whether to handle the message right now. If yes, write $x$ and $v$ into the local storage, updates its timestamp, sends $update(i, \tau_i, x, v)$ to other replicas also containing key $x$ and replies the acknowledgement to the client; if not, add this client message into the pendingClientMessages (Algorithm 15 and Algorithm 16).
Algorithm 13 handleGetMessage(cma)
1: gm ← cma.GetMessage;
2: if not handleMessagesNow(gm.timestampList) then
3:   pendingClientMessages.add(cma);
4: else
5:   Record rec ← handleGetMessages(gm);
6:   send reply containing rec to the source client;
7: end if

Algorithm 14 handleGetMessages(gm)
1: read from the local database and get the record;
2: return the record;

Algorithm 15 handlePutMessage(cma)
1: pm ← cma.PutMessage;
2: if not handleMessagesNow(pm.timestampList) then
3:   pendingClientMessages.add(cma);
4: else
5:   handlePutMessages(pm);
6:   send reply to the source client;
7: end if

Algorithm 16 handlePutMessages(pm)
1: insert the ⟨key, value⟩ into local database;
2: for Dependency dep: pm.timestampList do
3:   if dep.Edge.Vertex1 = serverId and AdMatrix[serverId−1][dep.Edge.Vertex2−1].contains(key) then
5:   else if timestamp[dep.Edge]<dep.Version then
7: end if
8: end for
9: construct a replicate message rm
10: for j ← 0; j<numOfServers; j ++ do
11:   if AdMatrix[serverId−1][j].contains(pm.Key) then
12:     send rm to server j + 1;
13: end if
14: end for
When the replica receives the timestamp message from the client, the replica needs to send its current timestamp and the replicated buckets to the client (Algorithm 17).

**Algorithm 17** handleTimestampMessage(cma)

1: send this replica’s current timestamp and replicated buckets to the source client;

When the replica receives the update timestamp message from the local client, the replica needs to update its timestamp after all causal dependencies are applied locally (Algorithm 18).

**Algorithm 18** handleUpdateTimestampMessage(cma)

1: if handleMessagesNow(cma.timestampList) then
2:   for Dependency dep: cma.timestampList do
3:     if dep.Version > timestamp[dep.Edge] then
5:     end if
6:   end for
7:   send reply to the source client;
8: else
9:   pendingClientMessages.add(cma);
10: end if

**Algorithm 19** handleMessagesNow(timestampList)

1: for Dependency dep: timestampList do
3:     return false;
4:   end if
5: end for
6: return true;

**Algorithm 20** handleServerMessage(ServerMessage sm)

1: if replicate message then
2:   handleReplicateMessage(sm);
3: end if
When the replica receives a replicate message from other replicas, it needs to check all pending updates in `pendingReplicateMessages`. For each update in `pendingReplicateMessages`, the replica checks whether the relevant causal dependencies of this update are applied locally or not. If yes, apply this update locally and remove it from the `pendingReplicateMessages`. If not, check next update in `pendingReplicateMessages`. This logic is achieved in Algorithm 21 and we will now discuss it. Function `handleReplicateMessage(sm)` implements predicate $J_2$ (step 3-20) and function $merge_3$ (step 21-27) mentioned in the proposed algorithm (Section 3.3). $intersectEdges$ is used to store edge $e$ where $e \in \tilde{E}_i \cap \tilde{E}_k$. Step 7-15 checks whether $\tau_i[e_{ki}] = \tau_k[e_{ki}] - 1$ or not. If $\tau_i[e_{ki}] < \tau_k[e_{ki}] - 1$, this update cannot be applied now, and check next update. If $\tau_i[e_{ki}] \geq \tau_k[e_{ki}]$, the update has already been applied locally, remove this update from the buffer. Otherwise, $\tau_i[e_{ki}] = \tau_k[e_{ki}] - 1$, the first condition in $J_2$ is satisfied and we use the flag $updateNow$ to represent this. $updateNow$ is true when $\tau_i[e_{ki}] = \tau_k[e_{ki}] - 1$ (step 14). Step 16-19 is used to check whether $\tau_i[e_{ji}] \geq \tau_k[e_{ji}]$ for each $e_{ji} \in \tilde{E}_i \cap \tilde{E}_k$, $j \neq k$. flag is false if one of the dependencies on edge $e_{ji}$ is not applied. Only when both conditions are satisfied, we will apply this update locally. Step 21-27 present how to apply the update by updating the version of edges in both $\tilde{E}_i$ and $\tilde{E}_k$.

Besides, after performing all executable updates, we need to check the pending client messages as well. Similarly, check each client message in `pendingClientMessages` and handle it if the causal dependencies are satisfied (Algorithm 21).
Algorithm 21 handleReplicateMessage(sm)

1: pendingReplicateMessages.add(sm.ReplicateMessage);
2: while each element rm in pendingReplicateMessages do
3:   for Dependency dep: rm.timestampList do
4:     if timestamp.contains(dep.Edge) then
5:       intersectEdges.add(dep.Edge);
6:       end if
7:     if dep.Edge.Vertex1 = rm.ServerId and dep.Edge.Vertex2 = serverId then
8:       if timestamp[dep.Edge] < dep.Version - 1 then
9:         break;
10:       else if timestamp[dep.Edge] ≥ dep.Version then
11:         remove this element from pendingReplicateMessages;
12:         break;
13:       else
14:         updateNow ← true;
15:       end if
16:     else if dep.Edge.Vertex2 = serverId and timestamp[dep.Edge] < dep.Version then
17:       flag ← false;
18:       break;
19:     end if
20:   end for
21:   if updateNow and flag then
22:     insert ⟨key, value⟩ into local database;
23:     for ⟨Edge, Long⟩ e: intersectEdges do
24:       timestamp[e.Key] ← max(timestamp[e.Key], e.Value);
25:       end for
26:     remove this element from pendingReplicateMessages;
27:   end if
28:   clear intersectEdges;
29: end while
30: handlePendingClientMessages();
Algorithm 22 handlePendingClientMessages()

1: while each element cma in pendingClientMessages do
2:   if get message then
3:     if handleMessagesNow(cma) then
4:       Record rec ← handleGetMessages(cma.GetMessage);
5:       send reply containing rec to the source client;
6:       remove this element from pendingClientMessages;
7:     end if
8:   else if put message then
9:     if handleMessagesNow(cma) then
10:    handlePutMessages(cma.PutMessage);
11:    send reply to the source client;
12:    remove this element from pendingClientMessages;
13:   end if
14:   else if update timestamp message then
15:     if handleMessagesNow(cma) then
16:       for Dependency dep: cma.timestampList do
17:         if dep.Version>timestamp[dep.Edge] then
19:         end if
20:       end for
21:     end if
22:   send reply to the source client;
23:   remove this element from pendingClientMessages;
24: end if
25: end while

4.2.4 MultiThreading

DKVF supports multiple threads. A new thread is spawned to handle the event when an event-handler is invoked. If a replica receives two PUT operations at the same time, there are two different threads to handle these operations. Multiple threads might read or write the shared variables (e.g., timestamp) at the same time, so it is necessary to add some restrictions (e.g. lock) to avoid mistakes. In our implementation, we use the ReentrantReadWriteLock which is an implementation of ReadWriteLock in JAVA to
allow concurrent reads but avoid concurrent writes. In the next chapter, we present some performance evaluation results.
In this section, we present the results obtained from experiments on the algorithm [38] for client-server architecture proposed in [1] and on our algorithm [36, 37] for peer-to-peer architecture proposed in Section 3.3.

**Figure 5.1:** An example for full replication.

**Figure 5.2:** An example for partition replication.
5.1 Experimental Setup

We consider three clusters (full replication (Figure 5.1), partial replication (Figure 4.2), and partitioned replication (Figure 5.2)). In the fully replicated cluster, each replica replicates all keys locally, and each client can access any replica. In the partitioned replicated cluster, replica 1 and 3 replicate keys mapped into bucket 0 and 1, and replica 2 and 4 replicate keys mapped into bucket 2 and 3. Each replica can also access any replica. We will also evaluate the partially replicated cluster introduced in Section 4.2. All replicas and clients are simulated in one machine with 16GB memory and 500GB storage. As for the variables in properties.txt, we set recordcount=100, operationcount=1000, readproportion=0.5, and insertproportion=0.5. We use one thread to simulate the client.

5.2 Improvement on Metadata Size

As we mentioned before, the protocol for peer-to-peer architecture adds fewer additional edges into the augmented share graph $\tilde{G}$. Hence, there are fewer augmented $(i, e_{jk})$-loops and fewer edges to track in some cases (e.g., the partial replication in Figure 4.2). Table 5.1 shows the differences among replica timestamps in two algorithms for the partially replicated cluster. The algorithm for peer-to-peer architecture requires replicas to track fewer edges to maintain the replica-centric causal consistency. We reduce the size of replica timestamp for implementing replica-centric causal consistency in a partially replicated system.

5.3 Improvement on Network Communication

We also implement some experiments in above clusters by using both algorithms and record the amount of PUT/GET operations and exchanged metadata (i.e., the
number of exchanged edges from the client to the replica). Since the procedure to issue and handle updates among replicas are identical in both algorithms, we don’t compare the exchanged metadata among replicas. Since the algorithm for peer-to-peer architecture reduces the replica timestamp, the exchanged edges among replicas are also fewer. In the fully replicated cluster and the partitioned replicated cluster, the replica timestamps are identical in both algorithms. For example, in the partitioned replication, \( \tau_1 = \tau_2 = \tau_3 = \tau_4 = \{e_{13}, e_{31}, e_{24}, e_{42}\} \) in both algorithms. The experimental results in the fully and partitioned replicated clusters are shown in Table 5.2 and Table 5.3 respectively. The column PUTs and GETs are the numbers of put and get operations issued by the client. Although we set read/write ratio to be 1:1 in properties.txt, the operations randomly generated by the YCSB don’t follow this ratio exactly. The ratio between put and get operations is roughly 1:1 but not identical in all experiments. The client timestamp is attached in both PUT and GET operations. Hence, the different read/write ratio doesn’t affect the induced conclusions much. According to the amount of exchanged edges and operations, we know the average number of exchanged edges per operation. For instance, in the partitioned replicated cluster, client 1 in the algorithm for peer-to-peer architecture attaches 4 (i.e., 4000/1000) edges with each operation to the local replica 1; and the amount of exchanged edges from client 1 in the algorithm for client-server architecture to all

<table>
<thead>
<tr>
<th></th>
<th>( r_1 )'s timestamp</th>
<th>( r_2 )'s timestamp</th>
<th>( r_3 )'s timestamp</th>
<th>( r_4 )'s timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer-to-peer</td>
<td>( e_{32}, e_{21}, e_{14}, e_{41}, e_{23}, e_{12} )</td>
<td>( e_{32}, e_{21}, e_{14}, e_{12} )</td>
<td>( e_{32}, e_{21}, e_{14}, e_{12} )</td>
<td>( e_{14}, e_{41} )</td>
</tr>
<tr>
<td>client-server</td>
<td>( e_{32}, e_{21}, e_{14}, e_{41}, e_{23}, e_{12} )</td>
<td>( e_{32}, e_{21}, e_{14}, e_{24}, e_{12} )</td>
<td>( e_{32}, e_{21}, e_{14}, e_{24}, e_{12} )</td>
<td>( e_{32}, e_{21}, e_{14}, e_{24}, e_{12} )</td>
</tr>
</tbody>
</table>
Table 5.2: Exchanged edges in the fully replicated cluster

<table>
<thead>
<tr>
<th></th>
<th>PUTs</th>
<th>GETs</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer-to-peer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>501</td>
<td>499</td>
<td>12000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c2</td>
<td>511</td>
<td>489</td>
<td>0</td>
<td>12000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c3</td>
<td>517</td>
<td>483</td>
<td>0</td>
<td>0</td>
<td>12000</td>
<td>0</td>
</tr>
<tr>
<td>c4</td>
<td>507</td>
<td>493</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12000</td>
</tr>
<tr>
<td>client-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>496</td>
<td>504</td>
<td>3108</td>
<td>3060</td>
<td>2940</td>
<td>2892</td>
</tr>
<tr>
<td>c2</td>
<td>492</td>
<td>508</td>
<td>2844</td>
<td>2952</td>
<td>3192</td>
<td>3012</td>
</tr>
<tr>
<td>c3</td>
<td>461</td>
<td>539</td>
<td>2664</td>
<td>2964</td>
<td>3036</td>
<td>3336</td>
</tr>
<tr>
<td>c4</td>
<td>519</td>
<td>481</td>
<td>2724</td>
<td>3036</td>
<td>3216</td>
<td>3024</td>
</tr>
</tbody>
</table>

replicas is also 4000. The average number of attached edges is also 4 which is exactly the length of every replica’s timestamp.

As Table 5.2 and Table 5.3 present, the client mainly accesses the local replica in the algorithm for peer-to-peer architecture, and the client evenly accesses all accessible replicas in the algorithm for client-server architecture. The reason is that the local replica has a higher priority than other replicas in the algorithm for peer-to-peer architecture. Such optimization can be also applied in the algorithm for client-server architecture. Our extension aims to the peer-to-peer architecture. The communication between the client and the local replica doesn’t occupy network resources. This reduces the amount of data exchanged in the network greatly.
Table 5.3: Exchanged edges in the partitioned replicated cluster

<table>
<thead>
<tr>
<th></th>
<th>PUTs</th>
<th>GETs</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer-to-peer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>494</td>
<td>506</td>
<td>4000</td>
<td>976</td>
<td>0</td>
<td>948</td>
</tr>
<tr>
<td>c2</td>
<td>486</td>
<td>514</td>
<td>1020</td>
<td>4000</td>
<td>1016</td>
<td>0</td>
</tr>
<tr>
<td>c3</td>
<td>512</td>
<td>488</td>
<td>0</td>
<td>936</td>
<td>4000</td>
<td>1024</td>
</tr>
<tr>
<td>c4</td>
<td>495</td>
<td>505</td>
<td>996</td>
<td>0</td>
<td>1008</td>
<td>4000</td>
</tr>
<tr>
<td>client-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>517</td>
<td>483</td>
<td>1084</td>
<td>932</td>
<td>1024</td>
<td>960</td>
</tr>
<tr>
<td>c2</td>
<td>495</td>
<td>505</td>
<td>924</td>
<td>984</td>
<td>1096</td>
<td>996</td>
</tr>
<tr>
<td>c3</td>
<td>521</td>
<td>479</td>
<td>1024</td>
<td>956</td>
<td>1044</td>
<td>976</td>
</tr>
<tr>
<td>c4</td>
<td>510</td>
<td>490</td>
<td>996</td>
<td>972</td>
<td>1060</td>
<td>972</td>
</tr>
</tbody>
</table>

As for the experiments in the partially replicated cluster (results shown in Table 5.4), the algorithm for peer-to-peer architecture has already reduced the size of some replicas’ timestamp. For client 1, the amount of exchanged edges is larger in the algorithm for peer-to-peer architecture. However, most (≈ 80%) edges are sent to the local replica, and the communication between the client and local replica doesn’t introduce network traffic. The amount of exchanged edges to remote replicas is much smaller in the algorithm for peer-to-peer architecture. For client 4, its timestamp includes edges in $\tau_1$ or $\tau_4$. Client 4’s timestamp is \{e_{32}, e_{21}, e_{14}, e_{41}, e_{23}, e_{12}\} for both algorithms. However, the exchanged edges are much fewer in the algorithm for peer-to-peer architecture. The reason is that the attached timestamp with requests issued by the client only contains edges in the destination replica’s timestamp. The algorithm for client-server architecture sends the entire client timestamp to the destination replica. Such optimization can be applied in the algorithm for client-server architecture as well. Overall, we reduce the metadata to exchange by accessing local replicas with a high priority, using smaller replica timestamps, and only attaching necessary edges when issuing requests.
Table 5.4: Exchanged edges in the partially replicated cluster

<table>
<thead>
<tr>
<th></th>
<th>PUTs</th>
<th>GETs</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>peer-to-peer</td>
<td>$c_1$</td>
<td>517</td>
<td>483</td>
<td>6000</td>
<td>-</td>
<td>1032</td>
</tr>
<tr>
<td></td>
<td>$c_2$</td>
<td>287</td>
<td>262</td>
<td>-</td>
<td>2196</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>$c_3$</td>
<td>132</td>
<td>107</td>
<td>-</td>
<td>-</td>
<td>956</td>
</tr>
<tr>
<td></td>
<td>$c_4$</td>
<td>387</td>
<td>371</td>
<td>1788</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>client-server</td>
<td>$c_1$</td>
<td>527</td>
<td>473</td>
<td>2160</td>
<td>-</td>
<td>1458</td>
</tr>
<tr>
<td></td>
<td>$c_2$</td>
<td>276</td>
<td>264</td>
<td>-</td>
<td>3240</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>$c_3$</td>
<td>133</td>
<td>125</td>
<td>-</td>
<td>-</td>
<td>1548</td>
</tr>
<tr>
<td></td>
<td>$c_4$</td>
<td>361</td>
<td>398</td>
<td>2490</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5.4 Experimental Results on Large Clusters

Besides, above clusters with only 4 peers seem to be not large enough. Hence, we consider another two large partially replicated clusters (Figure 5.3 and Figure 5.4). The numbers on solid lines represent the shared bucket IDs. For instance, replica 1 and 2 share bucket 1.

![Figure 5.3: A partially replicated cluster with 8 peers.](image)

The exchanged edges in these clusters with the same experimental setup are shown in Table 5.5 and Table 5.6. Since the client and replica are in the same peer, we focus on the network communications between the client and remote replicas. For instance, the edges sent from client 2 to replica 1 in the peer-to-peer architecture is much fewer in both clusters. The reason is that replica 1’s timestamp in the algorithm for peer-to-peer architecture (i.e., $\{e_{12}, e_{21}\}$ in both clusters) is much fewer than that in the algorithm for client-server architecture (i.e., $\{e_{12}, e_{21}, e_{23}, e_{32}, e_{34}, e_{43}, e_{45}, e_{54}, e_{56}, \ldots\}$).
Figure 5.4: A partially replicated cluster with 12 peers.

Table 5.5: Exchanged edges in the partially replicated cluster with 8 peers

<table>
<thead>
<tr>
<th>peer-to-peer</th>
<th>PUTs</th>
<th>GETs</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>137</td>
<td>80</td>
<td>508</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_2$</td>
<td>325</td>
<td>205</td>
<td>208</td>
<td>5580</td>
<td>-</td>
<td>-</td>
<td>2088</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_3$</td>
<td>110</td>
<td>80</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1936</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_4$</td>
<td>110</td>
<td>60</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1712</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_5$</td>
<td>123</td>
<td>72</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2024</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_6$</td>
<td>316</td>
<td>204</td>
<td>-</td>
<td>-</td>
<td>1776</td>
<td>-</td>
<td>-</td>
<td>6972</td>
<td>-</td>
<td>520</td>
</tr>
<tr>
<td>$c_7$</td>
<td>130</td>
<td>56</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>912</td>
<td>-</td>
</tr>
<tr>
<td>$c_8$</td>
<td>59</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>512</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>client-server</th>
<th>PUTs</th>
<th>GETs</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
<th>$r_6$</th>
<th>$r_7$</th>
<th>$r_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>139</td>
<td>103</td>
<td>3388</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_2$</td>
<td>317</td>
<td>315</td>
<td>3122</td>
<td>2338</td>
<td>-</td>
<td>-</td>
<td>3388</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_3$</td>
<td>112</td>
<td>134</td>
<td>-</td>
<td>-</td>
<td>3444</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_4$</td>
<td>111</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>3010</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_5$</td>
<td>130</td>
<td>137</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3738</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$c_6$</td>
<td>308</td>
<td>329</td>
<td>-</td>
<td>-</td>
<td>3136</td>
<td>-</td>
<td>4032</td>
<td>-</td>
<td>1750</td>
<td>-</td>
</tr>
<tr>
<td>$c_7$</td>
<td>127</td>
<td>121</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3472</td>
<td>-</td>
</tr>
<tr>
<td>$c_8$</td>
<td>52</td>
<td>65</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1638</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 5.6: Exchanged edges in the partially replicated cluster with 12 peers

<table>
<thead>
<tr>
<th>peer-to-peer</th>
<th>PUTs</th>
<th>GETs</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>145</td>
<td>80</td>
<td>508</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c2</td>
<td>333</td>
<td>214</td>
<td>182</td>
<td>1440</td>
<td>-</td>
<td>-</td>
<td>5588</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c3</td>
<td>111</td>
<td>74</td>
<td>-</td>
<td>-</td>
<td>4212</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c4</td>
<td>110</td>
<td>62</td>
<td>-</td>
<td>-</td>
<td>4480</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c5</td>
<td>125</td>
<td>91</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5852</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c6</td>
<td>309</td>
<td>188</td>
<td>-</td>
<td>-</td>
<td>3924</td>
<td>-</td>
<td>-</td>
<td>14586</td>
<td>-</td>
<td>-</td>
<td>2088</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c7</td>
<td>124</td>
<td>72</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4920</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c8</td>
<td>61</td>
<td>47</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2196</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c9</td>
<td>60</td>
<td>29</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>690</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c10</td>
<td>198</td>
<td>137</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3220</td>
<td>-</td>
<td>7260</td>
<td>-</td>
<td>1467</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c11</td>
<td>65</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>744</td>
<td>-</td>
</tr>
<tr>
<td>c12</td>
<td>61</td>
<td>36</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2160</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>client-server</th>
<th>PUTs</th>
<th>GETs</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>135</td>
<td>105</td>
<td>6240</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c2</td>
<td>343</td>
<td>290</td>
<td>5712</td>
<td>4676</td>
<td>-</td>
<td>-</td>
<td>7336</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c3</td>
<td>117</td>
<td>123</td>
<td>-</td>
<td>-</td>
<td>6240</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c4</td>
<td>114</td>
<td>113</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6356</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c5</td>
<td>130</td>
<td>145</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7700</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c6</td>
<td>315</td>
<td>313</td>
<td>-</td>
<td>-</td>
<td>6412</td>
<td>-</td>
<td>-</td>
<td>8036</td>
<td>-</td>
<td>3136</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c7</td>
<td>117</td>
<td>115</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6496</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c8</td>
<td>54</td>
<td>61</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2990</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c9</td>
<td>57</td>
<td>53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>660</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c10</td>
<td>175</td>
<td>221</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4760</td>
<td>-</td>
<td>4200</td>
<td>-</td>
<td>2128</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c11</td>
<td>64</td>
<td>54</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>708</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>c12</td>
<td>52</td>
<td>53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2730</td>
<td>-</td>
</tr>
</tbody>
</table>
\{e_{65}, e_{67}, e_{76}, e_{78}, e_{87}\} in the cluster with 8 peers, and \{e_{12}, e_{21}, e_{23}, e_{32}, e_{34}, e_{43}, e_{45}, e_{54}, e_{56}, e_{65}, e_{67}, e_{76}, e_{78}, e_{87}, e_{94}, e_{95}, e_{510}, e_{105}, e_{106}, e_{116}, e_{117}, e_{712}, e_{127}, e_{128}, e_{128}\} in the cluster with 12 peers). Although client 2 can access replica 1, 2 and 5, the edge $e_{15}$ is not added into the augmented share graph for peer-to-peer architecture.

Replica 1 can only propagate dependencies to replica 2.

![Figure 5.5: An augmented (2, $e_{76}$)-loop for client-server architecture in Figure 5.4.](image)

In addition to replica 1’s timestamp, each replica’s timestamp is much smaller in the algorithm for peer-to-peer architecture. In the cluster with 12 peers, replica 2’s timestamp contains $e_{76}$ in the algorithm for client-server architecture, but not in the algorithm for peer-to-peer architecture. In the algorithm for client-server architecture, there exists a loop in Figure 5.4 which contains $e_{76}$ (shown in Figure 5.5). The dotted line represents the dependency can be propagated by client’s accesses. An update $u$ on edge $e_{76}$ can be propagated through the path $r_7 \rightarrow r_8 \rightarrow r_3 \rightarrow r_2$ without affecting $r_1$, $r_5$ and $r_6$. Hence, replica 2 needs to track $e_{76}$ to maintain causal consistency in the algorithm for the client-server architecture. As for the algorithm for peer-to-peer
architecture, there also exist loops in Figure 5.4 containing $e_{76}$, and every loop containing $e_{76}$ includes $e_{105}$ as well. The shared bucket between replica 5 and 10 is 5, and bucket 5 is also replicated in replica 6. Therefore, any update on edge $e_{105}$ propagating the update on edge $e_{76}$ is also sent to replica 6 (illustrated in Figure 5.6). Replica 2 doesn’t need to track edge $e_{76}$ in the algorithm for the peer-to-peer architecture. These results also prove that the algorithm for peer-to-peer architecture reduces the metadata (i.e., timestamp) and is more suitable for peer-to-peer architecture.
Chapter 6

Discussion and Conclusion

In this chapter, we conclude the major distributions in our thesis and discuss the related research about metadata in partial replication and the future work to improve current implementations.

6.1 Conclusion

This thesis proposes a protocol that is more suitable for implementing replica-centric causal consistency for peer-to-peer architecture in partially replicated systems by extending the protocols proposed in [1]. We also present the algorithms to implement replica-centric causal consistency using the metadata which only track necessary and sufficient edges in the share graph $G$ in above proposed protocol. The proposed algorithm involves smaller metadata and it also reduces network usages compared with algorithms in [1].

6.2 Limitation and Future Work

There are also some limitations for the proposed algorithm. The proposed algorithm for peer-to-peer architecture forces the local replica to propagate dependencies and the client has to wait for the acknowledgement which represents the dependencies are applied at local replica. In some scenarios, this process may take a long time which increases the network latency of requests. Besides, this thesis only evaluates the
metadata size. There are lots of other performance metrics (e.g., latency, throughput) which are also significant in distributed systems. Above proposals are implemented in DKVF, we can also compare them with existing protocols in the future.
Bibliography


