Contention Management for Distributed Data Replication

Yousef Abushnagh

School of Computing Science

Newcastle University

A thesis submitted for the degree of

Doctor of Philosophy

July 2013

Abstract

Optimistic replication schemes provide distributed applications with access to shared data at lower latencies and greater availability. This is achieved by allowing clients to replicate shared data and execute actions locally. A consequence of this scheme raises issues regarding shared data consistency. Sometimes an action executed by a client may result in shared data that may conflict and, as a consequence, may conflict with subsequent actions that are caused by the conflicting action. This requires a client to rollback to the action that caused the conflicting data, and to execute some exception handling. This can be achieved by relying on the application layer to either ignore or handle shared data inconsistencies when they are discovered during the reconciliation phase of an optimistic protocol.

Inconsistency of shared data has an impact on the causality relationship across client actions. In protocol design, it is desirable to preserve the property of causality between different actions occurring across a distributed application. Without application level knowledge, we assume an action causes all the subsequent actions at the same client. With application knowledge, we can significantly ease the protocol burden of provisioning causal ordering, as we can identify which actions do not cause other actions (even if they precede them). This, in turn, makes possible the client's ability to rollback to past actions and to change them, without having to alter subsequent actions. Unfortunately, increased instances of application level causal relations between actions lead to a significant overhead in protocol. Therefore, minimizing the rollback associated with conflicting actions, while preserving causality, is seen as desirable for lower exception handling in the application layer. In this thesis, we present a framework that utilizes causality to create a scheduler that can inform a contention management scheme to reduce the rollback associated with the conflicting access of shared data. Our framework uses a backoff contention management scheme to provide causality preserving for those optimistic replication systems with high causality requirements, without the need for application layer knowledge. We present experiments which demonstrate that our framework reduces clients' rollback and, more importantly, that the overall throughput of the system is improved when the contention management is used with applications that require causality to be preserved across all actions.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Graham Morgan for his patience, guidance, and constant encouragement throughout my PhD studies. Dr. Morgan has been very helpful and I think that I will never forget this.

I would also like to thank my fellow PhD students who have helped in various ways during my PhD studies, in particular Craig Sharp and Matthew Brook.

Certainly, I would like to thank my family, especially my brother, because without his assistance and sacrifices I would not have been able to reach this point.

Finally, I would like to express my sincere thanks to my wife for her patience when I spent too much time working, and for her faith in my abilities.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Distributed System

A distributed system is a well-known and powerful computing paradigm [1]. Several definitions of a distributed system have been given in the literature and all of them have their individual focus. One common aspect shared by these definitions is that a distributed system has the property to always appear to users as a single highly capable computer. This thesis summarizes these definitions for simplicity and defines a distributed system as a system with more than one computer connected by a network to achieve a common task. All definitions of a distributed system lead to the following advantages of the system:

- Reliability: it is possible to share resources across several sites in a distributed system. Resources in this context are hardware, software, and/or data. This feature improves the accessibility of these resources at all times by allowing the computers to carry out computation even when failures occur. Reliability involves aspects such as availability and fault-tolerance.

- Performance: by sharing resources, any task can be portioned across the different computers, and such a configuration enables a distributed system to perform more work in the same amount of time, because many computations can be carried out collaboratively between computers in parallel.

- Scalability: the capacity of any conventional (centralized) system poses a limit for the system's maximum size [2]. Since a distributed system is not centralized, this restriction on the maximum size does not exist, meaning that it can easily add more recourses as needed to the system.

These are the most common advantages cited for distributed systems, and make a distributed system more powerful than a centralized system, as well as being good reasons for building a distributed system.

As communication channels over long distances are rather expensive and do not offer adequate bandwidth, remote resources are replicated by nearby users which enhance the performance and availability of those resources. In addition, placing resources at a single site may introduce a bottleneck for the communication network.

An important goal of a distributed system is the sharing of resources. Sharing was first introduced in the 1960s in time-sharing systems [3] in the form of a shared single computer resource between many users. Subsequently, it has been used in operating systems such as UNIX to enable processes to share system resources and devices, and more recently in database systems such as Oracle to enable applications to share objects.

1.2 Data Replication

With the increasing popularity of distributed systems, data access anytime and anywhere has become more and more feasible, and replication of data has become the most widely used approach in distributed systems. Data replication means that data is replicated across multiple computers. Replication is a technique to provide fault-tolerance or to improve performance. It provides fault-tolerance by replicating data in more computers, so if one replica fails the system continues working by switching to one of the replicas. Also, it maintains multiple replicas to provide protection against a single point failure. This replication improves performance by allowing access to local data instead of remote access, and thus improves response time and eliminates communication overheads.

Data replication has been used in different network environments. Previously, data replication was used in well-connected environment for the purpose of providing fast local access and higher reliability in the case of failure. Recently, in disconnected environments, data replication has been used for availability as well as for performance and reliability. Full details of replication along with the need for replication are given in section 2.2.

A special form of data replication is caching, which means a temporary store frequently accessed data on the client's machine. By storing data locally, a client can avoid many further contacts with the server. For example, browsers and proxy servers store previous responses from web servers to reduce the latency of fetching data from the web server, since information previously cached can always be reused. Caching may need to employ hoarding methods to predict and retrieve data that will be used by a client. Hoarding methods that have been proposed in the literature are discussed in the related work section.

1.3 Replication and Consistency

Data replication allows users to cooperate concurrently to achieve a common task and may also allow computers to remain continuously operational, even in the event of a temporary network failure. For example, in some systems, users could cooperatively interact by replicating shared data on their devices to perform work concurrently, and later merge the updates with each other. In other systems (mobile environments), computers are frequently disconnected, and users are allowed to replicate shared data on their laptops to proceed work and, when reconnected, they merge the updates with other computers.

Unfortunately, this feature poses one serious drawback to replication protocols and this is that it incurs overheads in the performance, as stated in [4][5]. Because of having multiple copies of shared data on different computers, updating one copy makes that copy different from the other. Therefore, replication introduces the problem of data consistency. Consider, for example, a banking system in which bank accounts are replicated in many different sites to improve availability and improve read performance. Suppose a joint bank account shared between two clients (client 1, client 2) is replicated at a different bank's branches. Initially, the balance of the account is £50 in all replicas. Because both clients could access the account concurrently, in the case of network partition, or when a communication link fails between some branches, an inconsistent view may occur. Imagine this scenario: client 1 withdraws £20 at the nearest ATM machine at one of the partitions and, at the same time, client 2 withdraws £35 at another partition. Since the replicas are unable to communicate with each other, the updates performed at each replica are not propagated between each other. As a result, an inconsistent view occurs, since the account can be seen as having different balances depending on which replica the read operation was made. In addition, the account becomes overdrawn, since a total withdrawal of £55 was made. This simple example demonstrates how the mutually consistent view of the balance can be lost when the bank accounts are subject to replication.

1.4 Consistency Models

Maintaining shared data consistency is a major challenge because different users have different requirements; in addition, different applications have different consistency models. This makes the problem of maintaining a mutually consistent view between replicas (ensuring that all copies have an identical value) is more difficult. Therefore, the challenge to research in this area is to formulate an efficient replicas control protocol that supports different distributed system applications.

There are several consistency models which have been found in the literature, and they differ according to various parameters such as network characteristics, update synchronization, and update conflict detection and resolution. These consistency models can only support strong consistency forms or weak consistency forms [6]. The strong consistency model prevents inconsistency between replicas by reducing their availability and therefore users do not observe any inconsistencies in the replicated data. Weak consistency provides high availability as a cost of consistency by allowing replicas to diverge from one another, but eventually they will converge. To ensure the consistency model is satisfied, a suitable protocol must be available that can provide consistency within the replicas. There are two main types for providing one of the consistency forms: pessimistic protocols (for strong consistency) and optimistic protocols (for weak consistency).

Pessimistic protocols ensure that shared data must be consistent at all times. With pessimistic protocols, users do not observe any inconsistency between shared data.

In case some replicas are unavailable (due to network failure), pessimistic protocols inhibit any updates to be performed in shared data until the failure is recovered. When an update operation is performed to share data in one replica, it is synchronized to all the other replicas and therefore all replicas have to reach agreement on the ordering of operations. Such synchronization takes a great deal of communication time, especially when replicas are spread across large distributed systems. Pessimistic protocols scale very poorly in large scale environments since the synchronization overheads between replicas is too expensive [7]. Pessimistic protocols are mainly used when fault-tolerance is the main goal [8]. Pessimistic protocols can be classified into:

- Passive replication: passive replication requires a primary replica; any updates to shared data must first be sent to the primary replica where it is processed. The primary then propagates the update to all other replicas. This approach introduces a single point of failure and a bottleneck. In the event that the primary replica fails, the other replicas elect a new primary, which takes over the role of the failed primary. This is done through a protocol.

- Active replication: unlike passive replication, an update to shared data can be performed at multiple replicas instead of only one. This property removes the single point of failure and bottleneck drawbacks, since if a replica fails the updates are still processed by the other replicas. Active replication requires all replicas to process the updates in an identical order.

In contrast to pessimistic protocols, optimistic protocols lower synchronization overheads and the ordering between replicas. In optimistic protocols, updates can be served as long as any single replica is accessible; this property provides a higher availability but may result in shared data to be inconsistent temporarily. However, it will be consistent eventually and improves response time, but clients may read stale data (out of date) when they read data locally, which might not reflect updates performed on other replicas. These protocols are suitable in systems where it is not necessary for all replicas to be identical in order for clients to carry out their work. Such type of protocols has been successfully used in environments where scalability and availability requirements are important (i.e. Usenet, DNS) [9].

Both types of protocols have somewhat complementary roles in shared data. Pessimistic protocols emphasize fault tolerance and consistency but are not scalable. Optimistic protocols emphasize scalability and availability but do not provide up to date views of data.

1.5 Causality and Correctness

Causality (or causal precedence relation) refers to the preservation of the causal relationship that holds between actions. It determines the sequence in which actions must be processed so that the cause action and the effect action appear in the correct order. For instance, if an action 1 ($A1$) occurs before an action 2 ($A2$) on the client-side, then we may say that $A2$ is caused by $A1$ or $A2$ depends on $A1$. A causally preserving protocol would enforce this relation at the server-side. In protocol design it is desirable to preserve the property of causality between different actions occurring across a distributed application [10].

Preserving causality across client actions has a significant impact on data replication. For instance, reconsider the example in section 1.3 of the replicated bank account at different bank branches. As demonstrated, the balance of the account is initially £50 in all replicas. Suppose client 1 performed an action $A1$ to deposits £50 at one replica and then subsequently performed another action $A2$ to withdraw

£75 at the same replica. We may identify that $A1$ and $A2$ are causally related and expect $A1$ to be received by all replicas of the account before $A2$ is received. If this is not the case, then some accounts may become overdrawn and therefore the bank charges an overdraft fee.

However, replication protocols cannot ensure by themselves the property of the causality, and so some dependency mechanism management should be added to the protocols. Exploiting semantic application knowledge is one way to maintain the causality relationship. Without application level knowledge, we consider an action to cause all subsequent actions at the same client. With application knowledge, we can significantly ease the protocol burden of provisioning causal ordering as we can identify which actions do not cause other actions (even if they precede them).

Maintaining causality is an additional check that increases the chances of non-progression. Consider the following example: assume a client carried out three actions $(A1, A2, A3)$ locally. Assume, due to state conflicts, that $A1$ is not committable at the server-side. This would mean that neither $A2$ nor $A3$ are committable as they depend on $A1$ (causal relationship). The server would have to inform the client and the client would have to rollback its actions somehow to reflect the server's decision. Therefore, chances of non-progression are increased the more requirements are placed on the causal relationship.

1.6 Contention management

Contention management policies aim to guarantee progression of a system at some level of fairness by making the right decision when conflict occurs. The choice of contention management policy impacts strongly on the throughput of the system. There are several contention management policies in the literature, and each policy has a corresponding application based on many factors such as maximum amount of data access, previous history of actions, or actions priority. However, there are many areas in which these policies are used.

In distributed multiple access, contention management policies aim to reduce collision and increase the utilization of the medium access. Since the medium is shared between all nodes, whenever more than one node simultaneously tries to access the medium to transmit data, data collision occurs and the nodes have to retransmit the data causing repeated collision and contention. The contention manager decreases collisions and resolves contention among competing nodes by adjusting the backoff mechanism of the conflicting nodes (see subsection 2.5.3.1).

In transactional systems, contention management resolves conflicts between transactions accessing the same shared data [11]. It occurs at the decision to determine which transaction should proceed, which transaction should wait or abort, and how transactions should wait. A number of contention managers implemented by various policies which suit various benchmarking criteria are described in subsection 2.5.3.2.

In optimistic approaches, the contention management may be viewed as the ability to derive the optimum schedule (e.g. one which maintains most causality, one which satisfies the most access requests). In this respect, contention management becomes an attribute of the reconciliation phase of the protocol. The application dependency occurs only in application defined causal definitions across shared data accesses, and decides what to do when accesses cannot be satisfied.

1.7 Contribution

In this thesis, we present a framework for contention management within optimistic replication schemes when causality requirements are high and must be preserved. We provide a flexible and efficient contention manager policy that provides most commits actions and maintains the most causality in optimistic replication schemes. Our novel approach is to utilise a degree of application knowledge found in the causal relationship between different data items to create a contention manager that improves the throughput while maintaining overall fairness.

The contribution of the thesis may be summarized as follows:

- Implementing a novel contention management scheme that identifies areas on the contention in the system. As the contention increases for given shared data, conflicting clients will be unable to apply any state changes to the system in order to allow non-conflicting clients to progress normally.
- Using a predictive scheme to provide the framework with the ability to predict the clients that the data items are likely to access in the future.
- Using the predictability of client's actions to maintain the causality requirements.
- Analyzing the performance of the framework by conducting a series of experiments. These experiments were created to determine the performance of our framework that may be appropriate for optimistic replication systems with highly causality requirements.

1.8 Publications

1. Abushnagh, Y., Brook, M., Sharp, C., Ushaw, G., and Morgan, G. Liana: A framework that utilises Causality to Schedule Contention Management across Networked Systems. The 11th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2012).

1.9 Thesis Outline

This thesis is organised as follows:

Chapter 2. We first introduce background studies related to optimistic replication systems. We then review the existing work related to such systems, classified into three categories: semantic reconciliation protocols, predictive protocols for data hording, and contention management policies.

Chapter 3. This chapter, presents a description of the framework (system model and the protocol design). This chapter also provides a generic example to demonstrate the implementation of the framework.

Chapter 4. We describe the design of the simulation and the implementation details to measure and compare the performance of the framework.

Chapter 5. In this chapter, we present performance results from a number of experiments which carried out. Each result is followed by an analysis to identify the characteristics of the framework.

Chapter 6. This chapter gives the conclusion of the study and suggests possible future work.

Chapter 2

Background and Related Work

This chapter contains the background studies that are related to optimistic replication systems, semantic-based reconciliation in optimistic replication systems, and maintaining causality in such systems. It begins by describing causality and how to implement it in distributed systems. Then, the chapter provides an overview of data replication along with the need for replication, and its environments. Next, we review the existing work of the popular optimistic replication systems in the context of their consistency models, application requirements and causality requirements. Finally, we justify our approach and describe the contribution of this thesis.

Throughout this chapter, we give several examples. To make these examples clear, we use a space time diagram in which we draw the operations of process, client, and replica along a time axis. The time axis is always drawn vertically with time increasing from up to down.

2.1 Maintaining the Causality Relationship

In a real-world scenario, it is taken for granted that things (events) happen according to the causality relation, which simply means that a cause event must happen before an effect event. However, in distributed systems, this may not be the case due to communication delay, and causality may be violated. Therefore, there must be a mechanism to handle the ordering of events, otherwise inconsistency and corrupted data may arise. The ordering of events is an important issue for system execution because it determines the operations behaviour that can be expected by the distributed applications [12].

Causality (or the causal precedence relation) refers to the preservation of the causal relationship that holds between events. Causality is based on the happened-before relation (see section 2.1.1), which means an event e may causally affect another event e' if, and only if, e precedes e'. Causality determines the sequence in which events must be processed so that the cause event and the effect event appear in the correct order. That is, if two events are causally related and have the same destination, they are delivered to the application in their sending order.

Causality helps solve a range of problems in distributed systems, such as the design of distributed algorithms, the tracking of dependent events and concurrency measures [12]. Also, causality plays an important role to maintain consistency in replicated data, for example, consider a shared directory replicated across three sites as illustrated in figure 2.1. Suppose that an update is executed on the local replica at each site, then propagated to the other sites and executed there in its original form upon its arrival. As shown in figure 2.1, update $u2$ is performed after the arrival of $u1$ from site 1 and, therefore, $u2$ may be dependent on $u1$. However, since $u2$ arrived and is executed before u1 at site 3, inconsistency may occur in the system due to a causality violation. For example, if $u1$ is to update a file in the shared directory, and $u2$ is to delete the same file, then the execution of $u2$ before $u1$ at site 3 will result in $u2$ referring to non-existent content. Therefore, it is important that all sites see causally related updates in the same order to maintain data consistency.

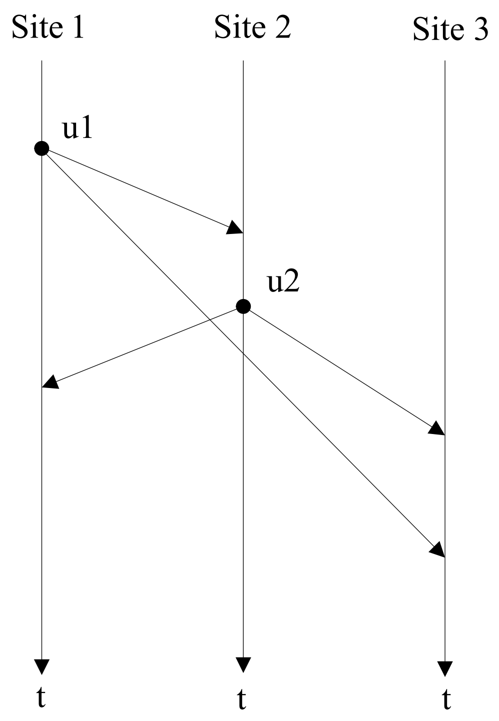

Figure 2.1: A Scenario for replicated shared data

Due to the importance of causality, several protocols [13][14][15] have been developed to capture the causality relation in parallel and distributed systems. These protocols use one of the following techniques [16]: causal histories, a Lamport timestamp, or a vector timestamp.

In this section, we present methods of maintaining the causal relationship between events. We start with an easy to understand concept of the happens-before relation, which is the base of causality, and then we continue to discuss how to maintain the causality.

2.1.1 The Happens-before Relation

Lamport [10] defined the notion of happens-before relation between events in a distributed system. The happens-before relation captures the notion of one event happening in the past of anoter. The expression $e \rightarrow e'$ is read e happens before

e'. The happens-before relation can be observed if one of the following conditions is true:

- C1: If e and e' are events in the same process and e occurs before e', then $e \rightarrow e'$.
- C2: If event e is the sending of a message by one process and event e' is the receiving of the same message by another process, then $e \rightarrow e'$.
- C3: If $e \rightarrow e'$ and $e' \rightarrow e$", then $e \rightarrow e$".

C1 states that the causality relation is preserved between events of the same process, and C2 captures the causality between events of different processes, while C3 states the transitive property.

2.1.2 Lamport Timestamp

Lamport invented a simple mechanism by which the happens-before relation can be maintained between events in the distributed system, called Lamport timestamps. The Lamport timestamp is a monotonically increasing counter, and to apply a timestamp to events, an event e assigns a time value $C(e)$ on which all processes agree. These time values must have the property that if $e \rightarrow e'$, then $C(e) < C(e')$. To maintain the happens-before relation, a timestamp C_i at process p_i is initially set to zero and advanced according to the following rules:

- R1: When an event occurs at process p_i, p_i increments its timestamp as follows: $Ci = Ci + 1$,
- R2: When a process p_i sends a message m, it piggybacks on m the value $t = Ci$,

- R3: On receiving (m, t), a process p_j set $Cj = max(Cj, t) + 1$.

Figure 2.2 shows an example of Lamport timestamp progress according to the above rules with three processes, each process with its own clock and an initial time value of zero.

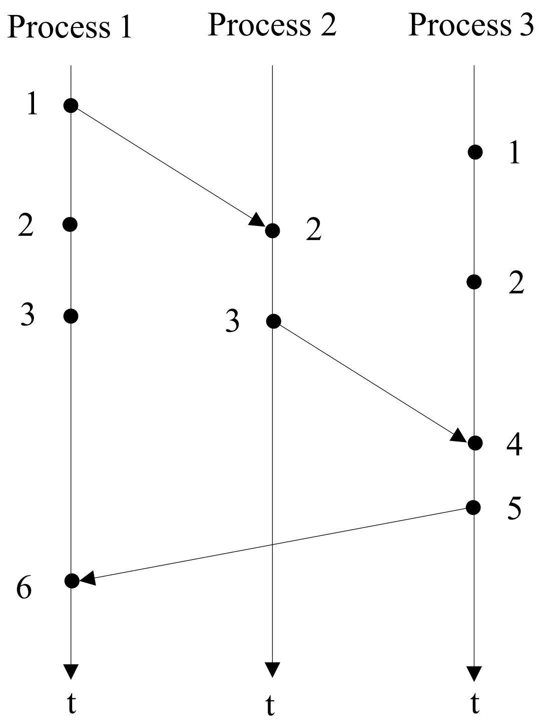

Figure 2.2: Lamport timestamp

2.1.3 Vector Clocks

Although Lamport states that if $C(e) < C(e')$ then event e precedes event e'. However, this does not necessarily imply that e and e' are causally related. For example, as shown in figure 2.2, the timestamp of the first event of $process1$ is less than the timestamp of the second event of $process3$, although they are not causally related. Furthermore, two events e and e' are concurrent, denoted by $e \parallel e'$, if neither e happened before e' nor e' happened before e. Consider figure 2.2 again, in which the third event of $process1$ and the second event of $process2$ have an identical timestamp. Events that are causally independent may get the same or different timestamps and it makes no difference in which order they occur.

A vector clock timestamp[17][18] provides a way to capture causality and concurrency between events. A vector timestamp $VC(e)$ assigned to an event e has the property that if $VC(e) < VC(e')$, then event e is known to causally precede event e'. In a vector clock, each process p_i keeps a vector V_i, which it uses to timestamp local events with the following properties:

- $Vi[i]$ is the number of events that have occurred so far at p_i.
- If $Vi[j] = k$ then p_i knows that k events have occurred at p_j.

In the vector clock timestamp, each process maintains a local n-element array $VCi[1,..,n]$, where n is the number of processes as shown in figure 2.3. $VCi[i]$ describes the logical time progress of p_i. $VCi[j]$ represents process p_i's latest knowledge of process p_j local time. The vector clock element is a non-negative integer number, initially set to zero. Each process p_i updates its VCi according to the following rules:

- R1: When an event occurs at process p_i, p_i increments its counter element of VCi as follows: $VCi[i] = VCi[i] + 1$,
- R2: When a process pi sends a message m, it piggybacks on m the value $vt = VCi$,
- R3: When process p_j receives (m, vt), p_j sets it logical time as follows: $VCj[k] = max(VCj[k], vt[k])$, then p_j increments its local element as follows: $VCj[j] = VCj[j] + 1$.

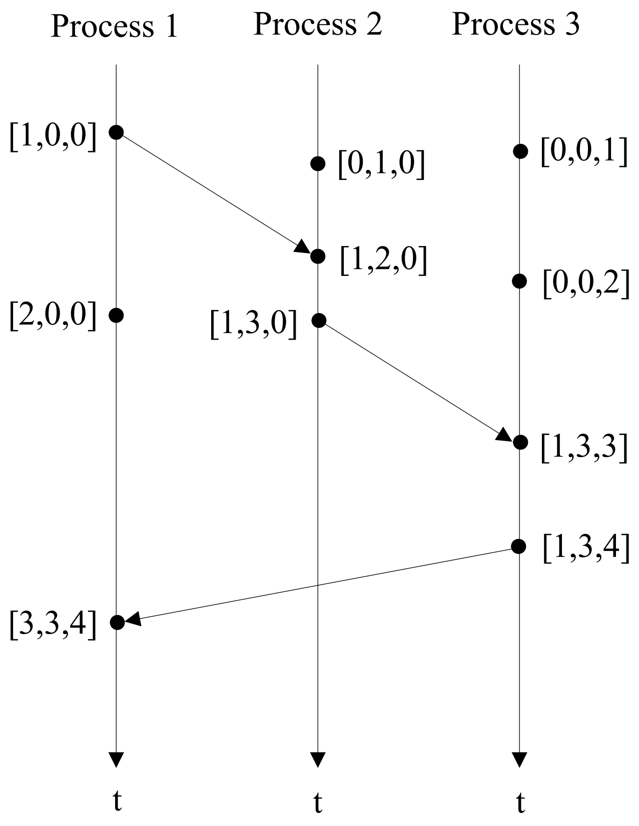

Figure 2.3: An example of vector clocks' timestamps

2.1.4 Directed Graph

A graph is a collection of vertices (or nodes) and a collection of edges (or arcs) that connect some pairs of vertices, where a vertex may be anything. Typically, a graph is used to model the relationship between two or more vertices from a certain collection. For example, the pages in a website and their hyperlinks can be modelled as a graph whose vertices are the pages available at the website and whose edges are the hyperlinks between these pages; page x to page y exists if and only if page x contains an edge to page y. A graph may be undirected, meaning that there is no distinction between two vertices associated with each edge, or directed in which all edges are assigned direction. A direct edge is an edge such that, one of its endpoints is designated as the tail and its other endpoints are designated as the head.

A directed graph [19] G is defined as an ordered pair (V, E) where $V(G) = \{$v1, v2, ..., vn $\}$is a set of elements are called vertices, and $E(G)$ is a set edges. Each edge is an ordered pair $(v1, v2)$ of vertices. In a directed graph, a path is a sequence of vertices connected by edges, for each edge $e = (v1, v2)$ is represented as $v1 \rightarrow v2$; $v1$ is called the tail of the edge and $v2$ is the head.

Furthermore, a directed graph comes in two forms: cyclic and acyclic. A directed graph is cyclic if some numbers of vertices are connected in a closed chain, such that a cyclic path begins with a vertex and follows a sequence of directed edges that eventually returns to the same vertex. For instance, the path $v1 \rightarrow v2 \rightarrow v3 \rightarrow v1$ is labelled as cyclic because we move from $v1$, $v2$, $v3$ then return to $v1$. Consider the graph of figure 2.4: the path (C, F, E, C) is a cycle as we move from C through F and E, then return to C.

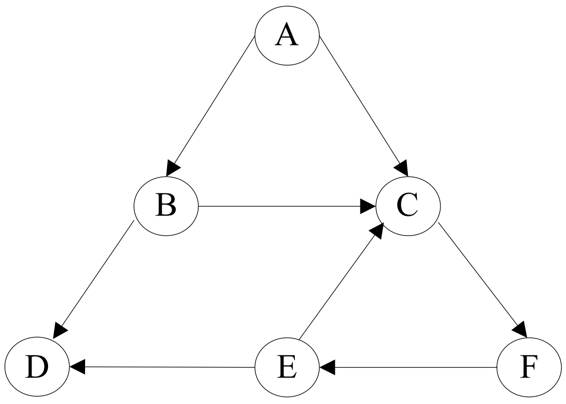

Figure 2.4: A simple directed graph

On the other hand, if there are no cycles, the directed graph is said to be acyclic. That is, a path is said to acyclic if no vertex appears more than once on the path. Consider the directed graph of figure 2.4 again; the path (A, B, D) is an acyclic path because no vertex appears twice. A directed acyclic graph can be used to represent the causal relation between a set of vertices [20], where a sequence of direct edges denote to a certain relationship that holds in pairs of vertices, recalling that the causality relation in section 2.1 can be represented by directed acyclic graph where events are vertices and where there is a direct edge from $v1 \rightarrow v2$ if, and only if, $v1$

precede $v2$. The directed acyclic graph has been used in data broadcast [21][22] for efficient broadcast scheduling in wireless mobile environments.

The previous section provides substantial information on different ways which can be used to order events and preserve the causality relationship between them in any distributed system (i.e. a distributed replication system). Next, we will describe replication systems and how weakly replication systems can take advantage of the causal ordering properties.

2.2 Shared Data Access

Replication of data in a distributed system means that several copies exist of the same data distributed over different computers. Data replication is used mainly for improving performance and availability reasons. Performance is improved by eliminating the communication overheads by enabling access to data locally instead of remotely. Availability is improved by allowing access to the data even when some of the replicas are unavailable or the communication between replicas is transient. Data replication is divided into two models, pessimistic and optimistic [23][24][8]. They represent the two extremes in the availability-consistency trade-off [25]. Pessimistic replication favours consistency over availability, while optimistic replication favours availability over consistency.

2.2.1 Pessimistic Shared Data Access

Pessimistic schemes provide strong consistency among replicas [7]. In these schemes, a user submits an update operation to some replica, before the operation is committed, the replica synchronises the operation to all other replicas. During the synchronisation, the version of updated data is unavailable to other users in order to ensure that executing the operation will not violate the consistency. When the replica receives confirmation that the update has reflected on all other replicas, it commits the update and sends the response to the user. In case some replicas are unavailable or communication fails, the synchronisation between replicas is blocked indefinitely, as is the response to the user [7].

The pessimistic approach is suitable in environments where connections between all nodes are robust and highly available. Therefore, the pessimistic scheme is not an option for environments where communication between nodes is intermittent or nodes are separated by a network partition.

2.2.2 Optimistic Shared Data Access

In contrast to the pessimistic scheme, optimistic schemes enable access to a replica without a priori synchronization with the other replicas and, therefore, allows distributed application to access any replica at any time even when there is network partitioning or when some replicas are unavailable. An update operation can be served as long as any single replica is accessible, and then updates are propagated in background between replicas. This property, however, has two significant implications. First, the states of replicas can be temporarily inconsistent; for example, an update can be applied to a single replica without the update being synchronously applied to other replicas. Second, concurrent updates to different replicas may introduce conflict. Despite these two drawbacks, the optimistic approach offers several advantages over the pessimistic, as follows [7]:

- Availability: as the replica does not need to wait for synchronization, applications can progress on their replicated data locally, even when a connection to the other replicas is temporarily unavailable.

- Scalability: optimistic schemes can support a larger number of replicas, because they require little synchronization between sites.
- Networking flexibility: optimistic protocols (i.e. epidemic protocols) work well over an intermittent connection by allowing updates to be exchanged between any pair of replicas.
- Site Autonomy: as the optimistic scheme requires little synchronization between sites, it supports asynchronous collaboration by allowing applications to work autonomously.

Optimistic approaches are used mainly for performance and availability in wide-areas such as distributed data services (i.e. Usenet, DNS) [9]. and mobile network applications such as Coda [26][27], Roam [28], Bayou [29][30], and IceCube [31][32].

2.3 Consistency Guarantee of Shared Data

A fundamental challenge of shared data access is to guarantee consistency among replicas, if one replica is updated the other replicas have to be identical. Different systems offer different guarantees and, therefore, consistency can be guaranteed according to a correctness criterion that is acceptable to applications of the system. It is worth mentioning that different applications can have different correctness criteria [33]. For example, a strong consistency guarantee is one of the correctness criteria for some applications which prevents any read or write operation going to the replicas in case of network partitions or network failure. This solution requires employing pessimistic techniques, which prevent inconsistencies between replicas by decreasing their availability.

When a network is partitioned or some replicas are unavailable, availability and consistency become an issue. As stated in [34], in the presence of network partitions, consistency and availability cannot be achieved at the same time. Therefore, the trade-off between availability and consistency offers another correctness criterion called the weak consistency guarantee (also called relaxed consistency). Some applications favour inconsistencies as a cost for improving availability, by accepting that replicas temporarily diverge, and then the system guarantees that eventually the replicas reach mutual consistency. Weak consistency has been successfully used in wide range of applications (i.e. mobile databases and collaborative software).

In the following subsections we introduce some consistency guarantees of shared data that have been investigated in the literature. We start by describing briefly the most restrictive guarantee which is known as strong consistency. Then, we introduce weaker guarantees that are more suitable for systems that require availability and scalability properties.

2.3.1 Strong Consistency Guarantee

The strong consistency guarantee means that the data must be consistent across all replicas at all times, and this requires that for any change to data all subsequent read operations performed on that data return the last value. In order to demonstrate the strong consistency guarantee in a concise way, we consider a virtual interleaving of the clients' operations (read or write) which reflect one possible correct execution as if a centralized system (unreplicated system) was used. The following strong consistency guarantees that will be described next take such virtual interleaving as a reference.

Linearizability is the strongest form of consistency guarantee, which requires that the order of operations in the interleaving is consistent with the real-time at which the operations were performed in the actual execution [3]. For instance, for any two operations O and O' occurring at times T and T' respectively, O occurs before O' in the interleaving order if, and only if, $T<T'$.

A weaker consistency guarantee than linearizability is sequential consistency, because it does not require that the interleaving operations order is consistent with the real-time at which operations are performed. However, the order of operations in interleaving is consistent with the program order in which each client executed them [3]. For example, for two operations O and O' performed at client Ci, O occurs before O' in the interleaving order if, and only if, Ci performed O before O'.

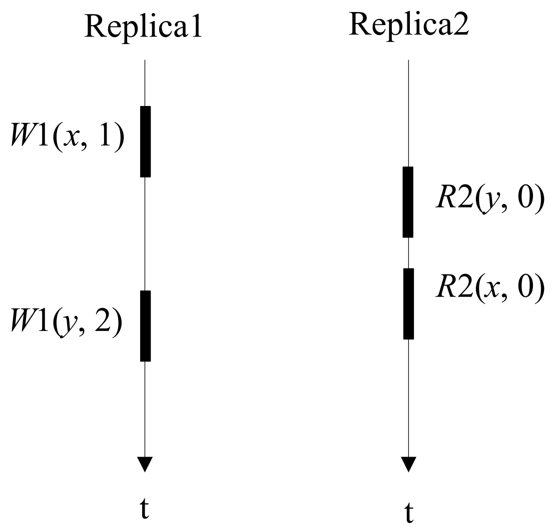

Figure 2.5: A replicated system that is sequentially consistent but not linearizable

Figure 2.5 shows an example of a replicated system that is sequentially consistent but not linearizable. In this example, two clients replicated the same data items x and y, and initially their values are 0. The example illustrates client1 updates data item x value to 1 by performing first the write operation $W1(x, 1)$, and later updateing data item y value to 2, also by performing write operation $W1(y, 2)$. Client2 performs two read operations $R2(y, 0)$ and $R2(x, 0)$ and sees the initial value for both

x and y. Consider the following interleaving order: $R2(y, 0)$, $R2(x, 0)$, $W1(x, 1)$, and $W1(y, 2)$. Obviously, the real-time requirement for linearizability is not satisfied, because $R2(x, 0)$ occurs after $W1(x, 1)$, but the interleaving order by which the clients performed is consistent with the sequential order.

Sequential consistency is similar to serializability.*"The main difference is that of granularity: sequential consistency is defined in terms of read and write operations, whereas serializability is defined in terms of transactions, which aggregate such operations"* [35].

The total consistency represents a weakening of sequential consistency, which requires that all operations are executed in all replicas in the same order. For example, given two replicas Ri and Rj, and two operations O and O' that are propagated to both replicas, if operation O is executed before O' at Ri, then Rj also executes O before O'. Figure 2.6 shows a replicated system with three replicas that ensure a total consistency guarantee between three operations $O1$, $O2$, and $O3$. As shown in figure 2.6 $O2$ is arrived at after $O1$ at replica1, while $O3$ is arrived at after $O1$ at replica2 and replica3; in this way, the total consistency guarantee system ensures that each replica makes the same ordering decision $(O1, O3, O2)$ to execute the operations.

A weaker consistency guarantee than total consistency is causal consistency, which requires that operations are executed according to causality relations (see section 2.1). If operation O' is caused or influenced by an earlier operation O, causal consistency guarantee systems require that every replica first executes O then O'. However, operations that are not causally related can be executed in any order on each replica.causal consistency guarantee is illustrated in figure 2.7, in a replicated system with three replicas. Suppose an operation $O1$ is executed in replica1. Later,

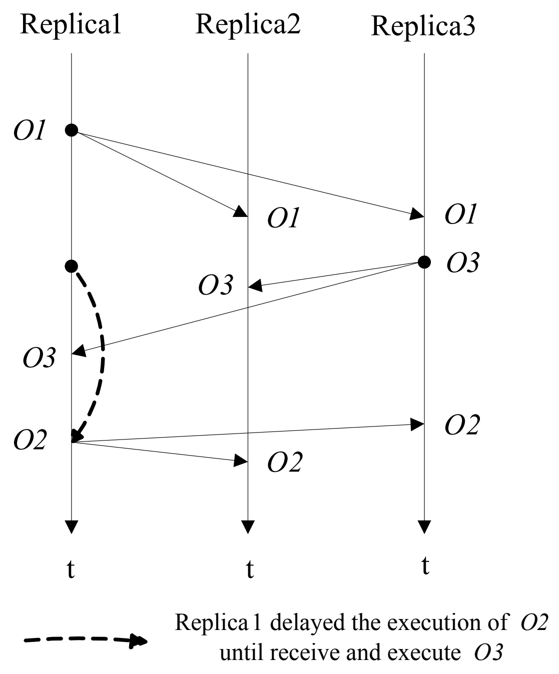
Replica1
Replica2
Replica3
O1
O1
O1
O3
O3
O3
O2
O2
O2
t
t
t
Replica 1 delayed the execution of O2 until receive and execute O3

Figure 2.6: Total consistency guarantee replicated system

an operation $O3$ is executed at replica3 after being received $O1$. In this scenario, $O1$ and $O3$ are potentially causally related because the computation of $O3$ may depend on $O1$ and, therefore, all replicas must execute them in the same order. Similarly, operations $O1$ and $O2$ are potentially causally related. Therefore, the replicated system is considered a causal consistency guarantee, because the causal ordering between operations $O1 \rightarrow O2$ and $O1 \rightarrow O3$ is maintained at the every replica.

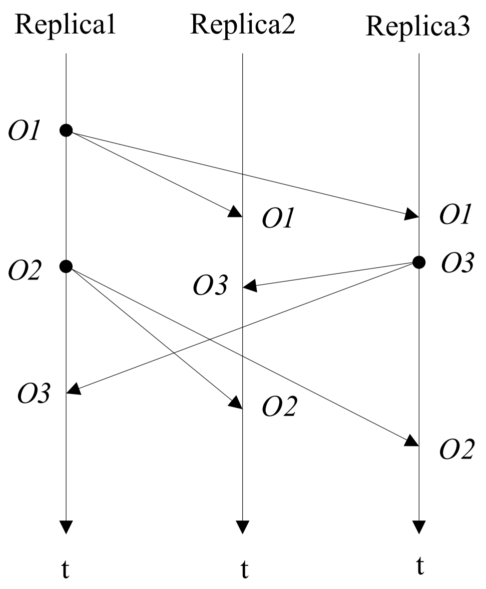

Figure 2.7: Causal consistency guarantee replicated system

A relaxing form of causal consistency guarantee is the FIFO consistency guarantee, which requires that operations performed by a client are executed on all replicas in the order in which they were performed, but operations from different clients are executed in a different order on other replicas [35].

Figure 2.8 illustrates the FIFO consistency guarantee system, in which the order between operations performed at each replica is maintained at all other replicas.

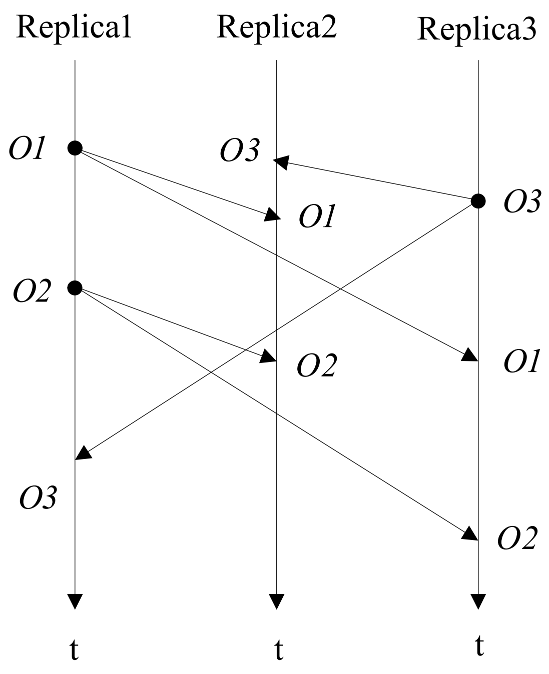

Figure 2.8: FIFO consistency guarantee replicated system

2.3.2 Weak Consistency Guarantee

In contrast to the strong consistency guarantee, weakening consistency lowers the ordering and reliability requriement placed on the underlying messaging protocol. A very weak consistency form called eventual consistency [36] guarantees the convergence of replicated states within an optimistic replication system. In principle, all replicas will converge, as past inconsistencies will be reconciled at some point during future execution. It follows that an absence of writes coupled with a window of full connectivity across replicas is required to ensure all replicas become mutually consistent. Two of the most popular, and well-known, distributed storage systems are Dynamo [37] and Cassandra [38]. These are examples of optimistic replication systems that utilise eventual consistency models, principally in achieving a high level of scalability.

Replicated systems that employ eventual consistency are allowed to temporarily diverge. Therefore, the eventual consistency guarantee differentiates operations into two phases. In the first phase, operations that are executed locally or on the nearest available replica are considered tentative. In the second phase, the operations are propagated and applied in some schedule with all other replicas in order to ensure a strong consistent view. In this phase, the operations become stable (or committed). The period between the first phase and the second phase is called the inconsistency window.

Figure 2.9 shows an example of an update made on a shared data item in a replicated system that guarantees eventual consistency. The figure illustrates three replicas that replicate a shared data item X. Initially, X is 0. A user at replica1 updates the value of X to 2 by performing write operation $W(X, 2)$, and later users at replica2 and replica3 read the value 0 from X (during the inconsistency window), which is the old value of X. Subsequently, the user at replica2 makes another read of X, and this time the latest value of X is returned.

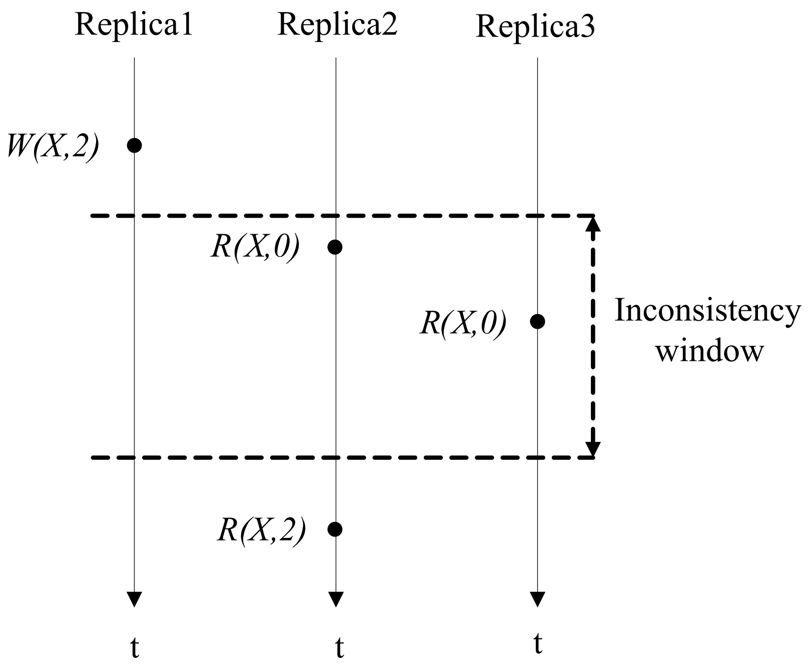

Figure 2.9: Eventual consistency guarantee replicated system

However, with the eventual consistency guarantee, replicas converge to an identical state, but cannot determine when have reached that state. As stated in [35], eventual consistent replicated systems work fine if clients always access the same replica. Sometimes problems may arise when the client operates on different replicas. Suppose a mobile user connects to a replica and performs some updates and is then disconnected. Sometime later, the same user moves to a different location and connects to a different replica. If the updates performed previously have not yet propagated, the user will see an inconsistent result. The notion of session guarantees [39] has been introduced to overcome this problem. Session guarantees provide an application with a view of the replicated data that is consistent with its reads and writes performed in session[1] even though these operations may be directed at

[1]A session is a sequence of read and write operations performed during the execution of an application.

different replicas. The session guarantees are explained as follows:

- Monotonic read consistency: if a client reads the value of a data item any subsequent read operation on that data item by the same client will always return that same value or a more recent value.
- Monotonic write consistency: a write operation by a client on a data item is performed before any subsequent write operation on that data item by the same client.
- Read your writes consistency: the effect of a write operation by a client on a data item will always be seen by a subsequent read operation on that data item by the same client.
- Writes follow reads consistency: a write operation by a client on a data item will be performed on the latest copy of that data item read by the same client.

However, allowing replicas to temporarily diverge implies that the replicated system and its applications need to consider the conflicts between updates and how they may be reconciled, and this is discussed in the next section.

2.4 Reconciliation of Shared Data

As we discussed in subsection 2.2.2, systems that utilize optimistic replication introduce temporary inconsistency by allowing operations to be executed on one single replica without requiring coordination with other replicas. These systems range from distributed file systems, such as Roam [28] and Coda [27], to distributed database systems, such as Bayou [29] and IceCube [32], and distributed storage systems, such as Dynamo [37] and Cassandra [38]. However, such systems raise the issues of conflict detection and resolution.

Reconciliation is a mechanism used in such systems to identify any conflicting updates made at different replicas to the same data item and resolve these conflicts in order to maintain the same global state between all replicas, according to the notion of eventual consistency criterion. Replica conflict occurs when two updates U and U' are applied to different replicas R and R' of the same data item d, if neither U nor U' has observed the effects of the other before executing.

Several works exist that attempt to achieve reconciliation of state. In recent years such works [29][31][32][40][41][42][43][44][45][46][47] have concentrated on the mobile environment, where transit connectivity has been such a reconciliation problem.

Reconciliation can be classified into syntactic and semantic techniques in terms of conflict detection and conflict resolution. The following subsections describe reconciliation techniques in more detail.

2.4.1 Semantic Conflict Detection

Semantic conflict detection uses application semantics to define and detect conflicts. Semantic conflict detection is based on operation preconditions, and conflicts occur when some operations do not satisfy their preconditions. For instance, for each operation that may potentially conflict with other operations, applications specify a precondition that must be evaluated as true for operations to be correctly applied, otherwise an operation indicates conflict. Systems like [29][30][31][32] are examples for using application semantics to detect update conflicts.

2.4.2 Syntactic Conflict Detection

In systems that rely on the syntactic technique, no explicit precondition is included by the application that performs the update operations. Instead, they depend on the timing of operations. The timestamp mechanism is employed with the syntactic-based technique in order to detect replica conflicts. Vector clocks accurately detect concurrent updates to the same data item [7]. Each replica carries an n-element array of timestamps $VCi[1, .., n]$, one for each replica that has a copy of the data item. Here $VCi[i]$ indicates that the last time an update to the data item was issued at replica i, and $VCi[j]$ indicates the last update to the data item issued at replica j that replica i has received (see subsection 2.1.3). Every time a data item is updated, a replica increments its vector clock element of the updated data item. During update propagation, replicas exchange their vector clock timestamps. Conflicts are detected between two replicas i and j as follows:

- If $VCi = VCj$, then the replicas have not been updated.
- If $VCi > VCj$, then the data item has been updated at replica i. Similarly, if $VCj > VCi$, then the data item has been updated at replica j.
- If none of the above is true, then there is an update conflict.

2.4.3 Conflict Resolution

When a conflict is detected, it needs be resolved to bring the replicated system to the identical consistent state. Resolving conflicts can be achieved manually or automatically. Manual resolution needs user intervention, whereas automatic resolution is application-specific. Consider the example of the calendar application, where a shared calendar is accessed by multiple users during network partitioning or perhaps some replicas are not accessible. A user requests a meeting proposing a time and location. If conflict is detected for conflicting reservations for meeting time slots, manual resolution may simply notify the user of the conflict and let the user decide how to resolve it. Coda [26] is an example that uses manual resolution of file conflicts. With automatic resolution, this is performed by an application-specific procedure which can try a different time slot. Bayou [29] is a system that supports such a method through a mechanism called merge procedures. Optimistic replicated systems favour automatic resolution since updates may propagate epidemically and the user may not be available when the conflict is detected.

The next section provides more details of optimistic systems approaches that aim to make use of the domain of the application to facilitate the process of handling irreconcilable differences and preserving causal dependencies across their shared state actions.

2.5 Related Work

This section surveys related work in the area of optimistic replication systems, hoarding approaches and contention management. First, we review some existing work of optimistic replication systems that attempts to implement semantic reconciliation to achieve eventual consistency. Then, we present work on hoarding, which is a strategy used in optimistic replication systems for retrieving necessary data likely to be used during a client disconnection. Finally, we review contention management policies, which are the mechanisms used in shared data systems to reduce conflicts and resolve contention between nodes.

2.5.1 Optimistic Replication Systems

2.5.1.1 Coda

Coda [26][27][48] is a distributed file system that supports disconnected operations and network partitions by allowing clients to replicate files and directories from servers. When a server is not accessible, a client reads and updates the contents of locally replicated files and directories.

A client in Coda operates in one of three states during its execution: hoarding, emulation, and reintegration. The hoarding state is normally when a client is connected to the server and operates on files and directories that are stored on the server. Also, a client in this state may replicate a certain set of files or directories in anticipation of disconnection. When disconnected, a client enters the emulation state and relies on replicated data. Upon reconnection, a client enters the reintegration state and propagates its updates in parallel to all servers, and then the client enters the hoarding state.

The emulation state may lead to replica divergence. Therefore, reconciliation is handled in the reintegration state. Coda uses both syntactic approaches (see section 2.4.2) to detect file conflicts and resolve them manually, while it uses semantic approaches (see section 2.4.1) to detect directory conflicts and resolve them automatically.

File conflict resolution is based on the version timestamp. Each replicated file is assigned a Coda Version Vector (CVV), with each element in CVV representing each server that stores that file. Each element counts the number of updates made at the server. A file that has been updated during disconnected replica will have a different CVV. Therefore, conflict updates can be detected by comparing the $CVVs$.

The state of replicas i and j are compared using their CVV. When every element of $CVVi$ is equal to the corresponding element of $CVVj$, the replicas are identical. If every element of $CVVi$ is greater than the corresponding element of $CVVj$, then the replica i has the later version of file; the contents of file are applied to replica

j and its $CVVj$ is modified to $CVVi$. The same is ture if $CVVj$ is greater than $CVVi$. If some elements of $CVVi$ are greater than but other elements are less than the corresponding element of $CVVj$, then conflict is detected. In this case the conflict is resolved manually by the client through the Coda repairing tool.

In contrast to files, directories rely on application knowledge. Each server maintains a resolution log along with each replicated directory. The resolution log records all actions made to the directory. Also, the resolution log contains all the necessary information to perform resolutions, such as names of new objects created and unique identifiers of all objects (created, deleted or modified). Conflict updates can be detected by simply comparing resolution logs. An example of such a conflict is when two disconnected clients each create different files on a common replicated directory. Coda easily solves this conflict be including both created files on a merged directory. Coda maintains the dependencies order between log entries from one server when logs are merged. For example, the entry for deleting file X must follow the entry of creating file X. Log entries from different servers can be merged in any order [49].

2.5.1.2 Bayou

Bayou [29][30] is an optimistic replication system designed with the goal of satisfying the consistency requirement of a shared state that resides across mobile devices. In essence, devices can access and update their local copy of the shared state while disconnected. Then, the device can synchronize with any other replica, propagating its updates and receiving new ones that it does not already know about.

Update propagation in Bayou is performed epidemically via an anti-entropy protocol, which means when a device connects to another replica it will send the updates it has made locally, but also receive any updates it does not already know about.

During this process, updates made at one replica will eventually be received by all other replicas. Updates are applied tentatively as they are received from other replicas. The order of tentative updates may vary at each replica. Within the system, one device is designated as the primary, which is responsible for deciding the final ordering of the updates and broadcasting this decision to all replicas. The ordering of the updates is based on the happens-before order achieved by the vector clock timestamp (see section 2.1.3). When a replica receives the decision, it will commit the updates, thus bringing it towards an eventual consistency. Committed updates are totally ordered (see subsection 2.3.2) according to the timestamp assigned by the primary node.

Conflict resolution in Bayou is handled by using both a dependency check and a merge procedure that are attached to each update action. The dependency check and merge procedure mechanism allows the application to specify how to detect conflicts and what steps should be taken to resolve these conflicts. Every time an action is executed, its dependency check is also executed. A dependency check consists of a query and its expected result; a conflict is detected if the query does not return the expected result. The merge procedure provides a number of alternate updates (determined by the application programmer) in case the original update cannot be succeeded. Once conflict is detected, a merge procedure is automatically executed and provides an alternate update.

2.5.1.3 IceCube

IceCube [31][32] goes beyond Bayou in creating a framework within which an optimal reconciliation of replica states may occur in the context of application dependencies. IceCube attempts to create a near optimal schedule in order to minimize local actions that cannot be honoured. Unlike Bayou, IceCube is considered a more flexible type of ordering that may violate the happens-before order. It relies on running actions in a different order because actions that are ordered by happens-before may be semantically commuting. IceCube proposes the notion of static and dynamic constraints to reduce the size of search space. Static constraints do not depend on the state of the replica and is used to determine the ordering in which actions are applied. Unlike static constraints, dynamic constraints depend on the state of replica and are used to verify the success or failure of an action. IceCube represents the relations between actions and constraints by a directed graph (see section 2.1.4), where nodes correspond to actions and edges are constraints.

In IceCube, an application can be one of the following two phases: the isolated execution phase or the reconciliation phase. During the isolated execution phase, a client performs actions against its local replica. Actions are recorded in the local log and remain tentative. The reconciliation phase starts when clients propagate their logs to a centralized schedule (in the server). The centralized schedule merges the logs of two or more clients to ensure a global consistent state between replicas. To this end, the reconciliation phase is divided into three stages: the scheduling stage, the simulation stage and the selection stage. The scheduling stage considers all possible combinations of merged actions to generate admissible schedules. The admissible schedules contain all possible orderings of actions that satisfy the static constraints. The simulation stage evaluates each admissible schedule against a shadow copy of the replica associated with it. A schedule that violates any dynamic constraints is aborted. The final outcome of this stage is one or more schedules. The selection stage chooses the best schedule according to application-specific criteria. The chosen schedule is the reconciled log, which can be applied to replicas to bring the system to one global state. Irreconcilable actions are dropped from the schedule and the system must handle these exceptions in an application dependent manner. Similar to Bayou, IceCube relies on a central node to decide the final optimal schedule between merged updates and then broadcasts this decision to all replicas.

APPA [50] has extended the IceCube to a fully distributed version. The motivation was to implement the IceCube reconciliation algorithm in a peer-to-peer data management system since a centralized algorithm is not suitable in such systems. This is because a central node may be a bottleneck. Furthermore, if the reconciler node fails, the whole replications system may be blocked until recovery. However, performance analysis of APPA's algorithm showed that there was not a significant improvement in reconciliation speed compared to IceCube's algorithm. This is because there are many dependencies that need to be maintained in a distributed reconciliation algorithm [51].

There are several other works, based on the aforementioned systems, which have also developed reconciliation to guarantee eventual consistency in optimistic replication systems. These systems use one of the following techniques: (i) timestamp ordering or (ii) application semantics. However, these works have one common goal, which is to achieve an efficient reconciliation that leads to an eventual consistent state.

2.5.2 Predictive Protocols

2.5.2.1 Seer

Seer [52][53] is a predictive hoarding system that supports disconnected operations in mobile environments. It infers semantic relationships between files based on a user's previous files access history. The main idea is that files are clustered using a measurement called semantic distance. These clusters are used to indicate the necessary files that should be hoarded to a mobile computer prior to disconnection. The notion of semantic distance defines relationships between files according to their distance. The smaller the semantic distance between files, the more they are closely related and, therefore, are probably involved in the same project; conversely, a bigger distance indicates that files are independent and belong to different projects. The semantic distance is measured based on the sequence of file reference (reference such as open or closed) rather than looking at the contents of file themselves. For example, consider the open and closed reference sequence of three files A, B, and C, such as $\{open\ A, open\ B, closed\ B, closed\ A, open\ C, closed\ C\}$, the semantic distance from open A to open B is 0 because A has not been closed before B is opened, whereas the semantic distance from open A to open C is 2. Similarly, the semantic distances open B to open C is 1.

Seer consists of two components, an observer and a correlator. The observer monitors the user behaviour and file access, classifying each access according to type and converting path names to an absolute format. The result of the observer is submitted to the correlator. The correlator calculates the semantic distance between various files to aid the system to assign each file to one or more cluster. Seer does not itself do the file hoarding; it relies on the replication system to perform the hoarding. Coda system [27] uses Seer during the hoarding state (described in section 2.5.1).

2.5.2.2 Spy Utility

Spy utility [54] is automated file hoarding that relies on program execution to build sets of related files. The main idea is to monitor file accesses in the background and recode these activities in the log. The background process analyzes the program executions in order to construct trees of various files accessed by each program.

Each node in the tree represents a file or an executable program. A node in the tree has branches of files or programs that have been opened or executed by the parent node. For example, an edge from node A to node B is added if program A calls program B, or program A uses the file B. After the trees have been built, the system unites all the trees for each specific program and records this in the database on the local disk. At hoarding time, the system presents this information to the user through a hoarding application tool. However, Spy utility automatically detects related files, requiring user intervention to select the sets to be hoarded.

2.5.3 Contention Management

2.5.3.1 Backoff Algorithms

In distributed multiple access environment, nodes are contend for the medium access for transmitting data packets in a distributed manner. Collision is considered a major problem in this environment, since the medium is shared between all nodes; whenever more than one node simultaneously tries to access the medium to transmit a packet, a packet collision occurs and the nodes have to retransmit the packets. If the collided nodes try to access the medium again, the packet will collide as the nodes may retransmit simultaneously.

Backoff algorithms are used in distributed environments to reduce collisions and resolve contention between competing nodes. Specifically, the Binary Exponential Backoff (BEB) algorithm that is used as a part of the Distributed Coordination Function (DCF) standardized by the IEEE 802.11 is widely used [55]. In the BEB algorithm, before transmission, a node senses the medium to determine whether it is idle. If the medium is idle, the node sets a timer to zero and creates a contention window with a size set to the minimum ($CWmin$). The node now selects a random interval between zero and $CWmin$. The node will then decrease this interval by one until it reaches zero. Once at zero, the node begins transmission. If a collision occurs, the node is backed off and it increases its timer by one and the contention window size is doubled, up to a ceiling value ($CWmax$). After each successful transmission, the timer is reset to zero and contention window size is reset to $CWmin$.

The performance of the BEB algorithm determines the contention window mechanism. For example, if the window size is too small and there are many nodes, then collisions are likely to occur. If the window size is too large and there are a few nodes with packets to transmit, then there will be unnecessary delays. In either case, the medium is not used efficiently. The BEB also suffers from a fairness problem where successful nodes will rest their contention window to minimum while other nodes can end up maintaining a large contention window, reducing their chance to access the medium to retransmit.

Many algorithms have been developed to enhance the BEB performance to efficiently utilize the medium. The Multiplicative Increase Linear Decrease (MILD) [56] was proposed to overcome the fairness problem in BEB. In the MILD, upon a collision, the collided nodes increase their contention window size multiplicatively by 1.5 and, upon successful transmission, nodes decrease their contention window size linearly by 1, instead of resting it to minimum. The algorithm has shown improvement to performance when the network load is high, but does not perform well when the active nodes' behaviour changes quickly from a period of high contention to low contention. This is because it cannot adjust its contention window quickly.

The algorithm Exponential Increase Exponential Decrease (EIED) [57] enhances the performance of the BEB. Whenever a collision occurs the contention window is doubled and is decreased by two upon a successful transmission. This algorithm provides better performance over BEB and MILD in terms of throughput. In the linear/Multiplicative Increase Linear Decrease (LMILD) algorithm [58], upon a collision, the collided nodes increase their contention windows by multiplying by two, while the nodes overhearing the collision increase their contention window linearly. Upon a successful transmission, all nodes decrease their contention window linearly. LMILD performs better than BEB and MILD in terms of throughput and fairness. The Pessimistic Linear Exponential Backoff (PLEB) algorithm [59] is a combination of the linear and exponential increment methods. The algorithm aims to merge the advantages of these two increment methods. Using exponential increments achieves high throughput by reducing the number of transmission failures, and using the linear increment reduces the average network delay. In this algorithm, whenever a collision occurs, the contention window is increased exponentially up to certain number then starts to increase linearly instead of exponentially incrementing. This method has shown improved performance over LMILD when used in moderately sized networks.

However, all these algorithms aim to optimise the size of the contention window and only differ by adjusting the contention window in a different manner.

2.5.3.2 Contention Managers

Deciding the order of shared data accesses to achieve the desired consistency model is a scheduling problem. Such ordering can satisfy a consistency model yet still exhibit a degree of data access bias that may cause problems in the application. Livelock is one extreme case but more common problems relate to prolonged periods of access starvation for processes, and this is an issue that increases in importance for real-time systems. Contention management is the additional attribute to a shared data access protocol that alleviates such bias from the scheduling of data access.

Contention management is an ordering technique used to resolve conflicts between transactions accessing shared data, and a conflict occurs if a transaction (victim transaction) is holding shared data which another transaction (attacker transaction) tries to access. Resolving such conflicts is managed by a contention manager. In transactional systems, the contention manager occurs at the decision to determine which transaction should be committed, which transaction should be aborted or delayed, and how the transaction should be delayed. An efficient contention manager makes the right decisions when conflict occurs. The choice of contention manager policy that guarantees a high level of progress and provides high throughput is not easy. This issue has resulted in a number of different contention manager policies [11][60]. The following contention manager policies have been described in the literature:

- Aggressive: the aggressive contention manager is very simple, as it always aborts the victim transaction.
- Randomized: the randomized policy randomly chooses aborting between conflicted transactions.
- KillBlocked: with the KillBlocked contention manager, upon a conflict the manager marks the attacker transaction as blocked. KillBlocked aborts the attacker transaction if: (i) it is marked as blocked many times, or (ii) its maximum waiting time has expired.
- Timestamp: the timestamp policy aims to provide fairness between conflicted transactions. The idea is to assign a timestamp at the beginning of each transaction. When a conflict occurs, the contention manager checks the attacker

transaction's timestamp, if it's timestamp is smaller the manager aborts it. Otherwise, the attacker transaction waits for fixed intervals, and in the middle of the intervals it will set the attacker transaction flag as defunct; if the flag is still set at the end of the intervals the attacker transaction aborted. A transaction timestamp resets when it is committed successfully.

- Greedy: this policy assigns a timestamp for each transaction when it starts. When conflict is detected, the contention manager aborts the transaction with a smaller timestamp.

- Polite: the Polite contention manager utilizes the exponential backoff technique to resolve conflict between transactions. When a conflict is detected, the attacker transaction backs off for a random amount of time within the range of 2^n where n is the number of times conflict has occurred. After a fixed number of attempts to access the same object, the Polite manger aborts the attacker transaction.

- Karma: the Karma contention manager attempts to always abort a transaction that has done less work. Karma keeps track of the number of objects opened by a transaction and its priority. It increments this priority with each object opened and resets to zero when a transaction commits. Priorities do not reset to zero if a transaction aborts. When a conflict is detected between transactions, and the attacker's transaction has lower priority, it aborts. Otherwise, it backs off for a fixed period of time and retries. When the number of retries exceed the difference in priorities between the attacker and victim transaction, the attacker transaction is aborted.

- Eruption: this policy is an extension of Karma. It is similar to karma, but when conflicts occur between transactions, it adds the attacker's transaction priority to the victim's transaction priority and then the attacker transaction backs off. The idea behind the extension is to increase the priority of the transaction that blocks multiple transactions and therefore finish quickly.

- Polka: the Polka contention manager is a combination of two policies, Polite and Karma. It combines the priority policies of Karma with the exponential random backoff of Polite. Polka works in a similar way to Karma but instead of backing off for a fixed number of intervals, it backs off exponentially; after a set amount of maximum backoff, the contention manager aborts the attacker transaction.

However, an evaluation of these policies in the literature [60][61] showed that no contention manager policy seems to work universally better in all circumstances.

2.6 Strong Causality

As mentioned in section 2.1, the principle of causality usually means the relationship between a set of actions (cause) and (the effect). We proceed in this section to establish the strong causality: strong causality specify that all possible effect actions can only be executed if, and only if, the cause action executed successfully [31]. This is because an inability to satisfy action that resulted in a conflict in shared state means that subsequent actions carried out by a client must be void. This is useful when the cause action produces some effect used by the effect action. Suppose that the client wants to copy a file after updating it. The update action must be a predecessor of the action that copies the file. Strong causality works for those systems that must maintain causality from the client's view. That is, a server must process all client requests in the order they were sent by a client. This is the type of system we are primarily concerned with.

2.7 Discussion

Achieving an implementation of an optimistic replication system that enforces eventual consistency balances the requirements of consistency, availability and timeliness. Increasing consistency (moving towards a more pessimistic approach) contradicts availability and timeliness. However, increasing inconsistency places a greater burden on the application as exceptions that are the result of irreconcilable differences in the shared state must be handled.

The amount of work required by an application to retrospectively handle exceptions increases as the need to maintain causality across client actions on the shared state increases. This is because an inability to satisfy an action that resulted in an irreconcilable difference in the shared state means that subsequent actions carried out by a client may also be void. However, earlier academic works like Coda, Bayou and IceCube (recall section 2.5) do attempt to maintain a degree of causality, but only at the application programmer's discretion (which would not be known explicitly in our approach as we expect client machines to not hold such information and so cannot use it in their messaging protocol).

Coda works very well in environments where conflicts are very rare and it is effective for shared objects with simple dependencies such as directories. Bayou and IceCube rely on the application programmer to specify the extent of causality, preventing a total rollback and restart. This has the advantage of exploiting application domains to improve availability and timeliness, but does complicate the programming of such systems as the boundary between protocol and application overlap. If the causality requirements of an application span many actions then the complication of injecting application specific exception handling will increase. In some instances, such exception handling defaults to an undesirable rollback scenario as ignoring irreconcilable actions or attempting alternative actions may not be viable. This is the worst case scenario for applications with strong causality requirements as the reconciliation of optimistic protocols does nothing but increase overheads, both in terms of throughput and the complexity of the programming model. In fact, the model becomes transactional in nature.

Transactions may offer a solution, but do carry a substantial overhead in their commit phase and are traditionally only employed at the end of a series of non-transactional accesses (e.g. for financial transfers). In addition, transactional systems provide schedules that do not exploit semantic properties that could provide a weaker, but still correct, consistency model (as in some optimistic approaches).

Contention managers are primarily related to transactional systems, where rollback is expected. However, because compensation is expected in eventually consistent protocols there has not been too much work in contention management in eventual consistency. However, in those cases in eventually consistent protocols where causality must be maintained, contention managers play an important role. In optimistic approaches, the contention manager may be viewed as the ability to derive the most optimum schedule (e.g. one which maintains most causality, one which commits the most actions). In this respect, contention management becomes an attribute of the reconciliation phase of the protocol.

Therefore, we need to develop a contention manager for an eventually consistent protocol that may require stronger causal guarantees. We require a contention manager that operates in a transactional like environment (as clients rollback in the presence of irreconcilable data accesses), but unlike transactions that can make use of application level semantic knowledge (probability of future causal relations), similar to optimistic approaches. However, such semantic knowledge is only maintained at the server side, meaning we want a contention manager that does not rely on the client's understanding of how to best exploit such knowledge (similar to transactions as clients being told to simply rollback or continue).

Maintaining semantic knowledge at the server side has the advantage of reducing the client messaging protocol overhead on both message size and semantic knowledge. In addition, removes the complexity of the application developer by delegating the handling of semantic knowledge and causality at the server.

2.8 Contribution Made by the Thesis

In this thesis, we present a framework which provides contention management for improving throughput when clients progress independently, but which also requires the property of eventual consistency in the presence of irreconcilable differences of state shared across clients. In addition, our framework is specifically designed for environments where preserving causality in the same manner as a transactional type system is a requirement.

We use causality inherent in the application layer to our advantage, in that we assume a degree of predictability in a client's actions. For example, in e-commerce, such predictability is commonly used when suggesting subsequent purchases to clients; for example, Amazon's website provides information regarding the popularity of items and the relationship between such items with regards to previous sales to improve shoppers' awareness of products. This may appear to a shopper as: "people who bought this item also purchased the following items $\cdots\cdot$". We use predictability in two ways:

1. Manage contention for popular items of shared data via a backoff scheme, and
2. Pre-emptively update a client's shared state of the data items they are likely to access in the future.

However, Backoff-based contention management has been used in distributed multiple access environment (see subsection 2.5.3.1) to reduce collisions and resolve contention between competing nodes and in software transactional memory (e.g. [60][62]), to resolve conflicts and avoid livelock and starvations when a conflicting transaction aborts and restarts. Backoff has also been demonstrated in message ordering (e.g. [63]) to attain replication systems with fault-tolerant properties. To the best of our knowledge, this is the first time it has been combined with the predictability of causality within the application domain in an optimistic manner.

We stress that it is the contention management aspect of our work that is the main contribution. To test our contention management approach, we have to construct a more complete protocol to fulfil our requirements. We believe that any protocol suited to satisfying our approach requirement (will be formalised in the next chapter) could be substituted to test our contention management framework.

2.9 Summary

This chapter has provided the necessary background relevant to the thesis. It started with the fundamental concept of causality and described the techniques that have been found in the literature to track causality in distributed systems. Then, the data replication and the difference between optimistic replication and pessimistic replication models were described. The consistency guarantee of the pessimistic replication model was briefly presented. More detail on the consistency guarantee of the optimistic model was discussed, specifically eventual consistency, since the thesis falls into eventual consistency. A description of achieving consistency within optimistic approaches was introduced and the ways to achieve eventual consistency was described. Next, an overview of several approaches related to optimistic replication, and how such approaches handle irreconcilable differences, were presented. Finally, we sum up the contribution of the thesis.

Chapter 3

Framework

In this chapter, we present our framework, the formal model and the protocol design. We start by describing the system model and we detail the system architecture and the requirements. Then we present the proposed protocol that is suitable for optimistic replication systems. Next, we explain client and server algorithms using pseudo-code. Finally, we introduce the contention management that will be augmented with the proposed protocol.

3.1 The Requirement

This section identifies the requirement of system to enable our contention management framework to be used in resolving conflict and maintaining causality, the system must fulfil the following requirements:

- Client/Server architecture.
- Clients adopting full replication.
- Utilizing eventual consistency correctness criteria.
- Adopting Pull-base update propagation technique.

3.2 Notations

In order to facilitate the tracking of the meaning of the symbols that will be used throughout this chapter, Table 3.2 provides a summary of these symbols.

Table 3.1: Summary of notations

Symbol(s)	Meaning
C	Client
S	Server
Ds	Data set
di	Data item
A, B, ..., etc	Refer to data items
m	Message
G	Graph represents the relations between data items
VV	Volatility value represents how many data items have been accessed
UMR	Update message request sent from a client to a server
MMR	Missed message request sent from a server to a client
$EARM$	Enhanced authoritative rollback message sent from a server to a client
DQ	Delta queue

3.3 System Model

The system used in this thesis is a typical distributed system consisting of a single server and a number of clients, as illustrated in figure 3.1. The server maintains all shared data and clients maintain duplications of such data. Clients communicate with the server through message passing. A single client is not aware of any other clients and can only communicate directly with the server. The system is asynchronous, i.e. different clients may run at different execution speeds, clients proceed sending immediately after they have submitted their message to server and the delay of messages is not bounded, possibly varying from one message to the next. However, there is no reason why the system cannot be extended to peer-to-peer (assuming each client holds shared data with epidemic message propagation as in Bayou). Communication channels between clients and server exhibit FIFO (first in first out) qualities but may lose messages.

Clients carry out actions locally on a duplication of the server state and periodically inform the server of their shared data accesses. A server receives these access notifications from clients and attempts to carry out all client actions on the master copy of the shared state. However, if this is not possible due to irreconcilable actions then clients are informed. On learning that a previous action was not achieved at the server, a client rolls back to the point of execution where this action took place and resumes execution from this point. Subsequent actions from such a restart may not be the same (the system is dynamic in this sense).

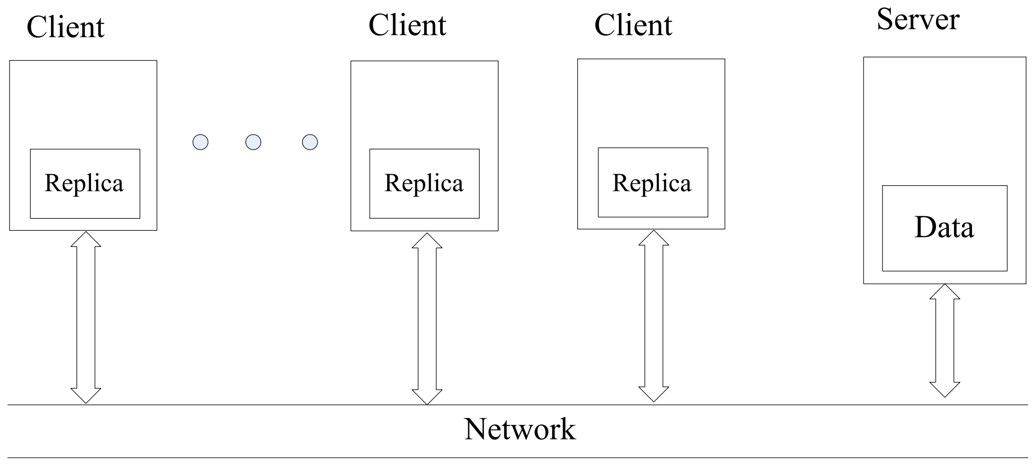

Figure 3.1: System Model

3.3.1 Data

Since replication is relevant to a distributed system, as explained in section 2.2, our model considers shared data access. We define the shared data between the server and each client as a collection of objects $\{d1,d2,\cdots,di\ \}$called a data set DS. We refer to a single object $di \in DS$ as a data item. Each data item di has a state and a logical clock value associated with it. Only the server can advance the logical clock for a data item.

3.3.2 Clients

Clients are denoted by $C1, C2, C3, \cdots, Cn$. Each client maintains a local replica of the data set maintained by the server (the master copy), as shown in figure 3.1. All client actions performed on shared data are directed only to their local replica. We define an action as an event invoked by the client that updates or reads a data item's state. An action may access at most a single data item at a time. Each client uses a number of logical clocks to aid in managing their rollback and execution:

- Client data item clock (CDI): exists for each data item and identifies the current version of a data item's state held by a client. This allows the server to recognise when a client's view of a data item is out of date. This is updated when the server informs a client when their actions were rejected for operating on stale data.

- Client session clock (CSC): clients increment each time they are requested to rollback. This allows the server to ignore messages belonging to out of date session.

- Client action clock (CAC): clients increment each time they carry out an action. This allows the server to recognise missing messages from clients.

The result of an action that accesses a data item's value in the local replica state results in a message that is sent to the server. This message contains the data item state, the CDI of the data item, the CSC, and the CAC. An execution log is maintained and all client messages belonging to actions that result in shared state accesses are added to it. This log aids the client rollback procedure.

A message arriving from the server indicates that a previous action, say An, carried out by a client was not possible or client messages are missing. All server messages contain a session identifier. If this identifier is the same, or lower, than the client's CSC, then the server message is ignored (as the client has already rolled back, and the server may send out multiple copies of the rollback message). However, if the session identifier is higher than the client's CSC, the client must update their own CSC to match it and rollback.

If the server sent the message due to missed client messages then only an action clock identifier and session identifier will be present (we call this the *missed message request*). On receiving this type of message, the client should rollback to the action point using their execution log. However, if the server sent the message because a prior client action was not possible then such a message will contain the latest state of the data item that An operated on, and the new logical clock value the client should install for this data item (we call this the *irreconcilable message request*). On receiving such a message the client halts execution and rolls back to attempt execution again from An.

Although a client will have to rollback when requested by the server, the receiving of a server message also informs the client that all their actions prior to An were successful (those with a lower CAC in the execution log). As such, the client can reduce the size of their execution log to reflect this.

3.3.3 Server

Our model consists of a single server, S, which maintains the master copy of the data set for the system, labelled DSs. From this data set the clients create their local replicas. The role of the server is to ensure that causal relationship between a client's actions is preserved and client replicas are eventually consistent.

Over time the state of data items maintained by a client will become inconsistent with those held by the server. This is a result of actions applied by other clients. As such, in order to ensure that updates are not lost and each client operates with the most up to date version of the data item, the server maintains a number of logical clocks to aid in informing clients when to rollback:

- Session identifier (SI): this is the server's view of a client's CSC. Therefore, the server maintains an SI for each client. This is used to disregard messages from an out of date session from clients. The SI is updated by one each time a client is requested to rollback.

- Action clock (AC): this is the server's view of a client's CAC. Therefore, the server maintains an AC for each client. This is used to identify missing messages from a client. Every action honoured by the server on behalf of the client results in the AC of that client being set to the CAC belonging to the client.

- Logical clock (LC): this represents the version number of a data item. Therefore, the server maintains an LC for each data item. The server uses this to inform clients of their CDI values. Whenever an action successfully accesses a shared data item, the LC of that data item is incremented by one.

A message from a client, say $C1$, may not be honoured by the server due to one of the following reasons:

- *Stale session* : SI belonging to $C1$ is greater than the CSC in $C1$'s message

- *Lost messages* : CAC in $C1$'s message is two or greater than AC

- *Stale data* : LC is greater than CDI in $C1$'s message

When the server has to reject a client's ($C1$) message, a rollback message is sent to the client.

3.4 System Protocol

We describe our protocol as a pair of algorithms, one for the clients' side and one for the server side. We first provide an overview of the protocol and then present the algorithms for each side as pseudo-code. The algorithms described can be seen in Algorithm 1 and Algorithm 2.

3.4.1 Overview

Our protocol consists of two parts: a client side algorithm and a server side algorithm. Each client executes the same algorithm in which, for each action issued, the result is communicated to the server.

A client is able to independently make changes to the data set. Once a single data item has been modified, the client algorithm must be executed before another action can be made. The algorithm does not block the client after having sent a message; the client is free to issue a new action modifying the state of a data item locally. Before a message can be sent, however, the client algorithm must process any messages that may have been received from the server. Message delivery is considered reliable but may lose messages.

While not presented in the algorithms, we assume that each client initially connects to the server to download a copy of the data set before being able to send a message to the server. When retrieving a copy of the data set, the latest version will always be provided.

3.4.2 Pseudo-code

Upon receipt of a message, the communication layer will place the message in a queue called *messages* for the process to access. Messages in the queue are processed in a FIFO order. We assume the primitives **add**, **remove** and **send** for use when manipulating collections in the pseudo-code.

Two message types exist in the system; a Request message is sent from client to server, and a Response message is sent from server to client in the event of a previous action carried out by a client was irreconcilable or a client message was missing.

When the client wishes to send a message, an update message request (UMR) entity is created and populated with the required fields. These fields are the data item state, client data item clock, client session clock and client action clock. The format of the clock is an integer, starting at zero, which is incremented by one when required.

A missed message request (MMR) entity is created and populated with the required fields when a client's UMR is missed. These fields contain only an action clock and session identifier. An authoritative rollback message (*ARM*) entity is created and populated with the required fields when a message needs to be sent to the client due to irreconcilable message. The fields of the entity contain the server's version of the data item (state, logical clock, session identifier and action clock) for the conflicted data item.

3.5 Protocol Description

As mentioned in the preceding section our protocol consists of two algorithms, a client side algorithm and a server side algorithm. This section provides a full description for both algorithms and illustrates them as pseudo-code.

3.5.1 Client Side Algorithm

The client executes the Algorithm 1 after having made an update to a data item. The client provides the new state for the data item modified as a parameter. Any messages received from the server are placed in the *messages* list. Before a client can send a new message, the list must contain no Response messages (no messages have been received from the server since the last message was sent).

If a Response message has been received, then this must be processed before the sending of the new messages. If the session identifier (SI) in the Response message is the same or less than the client's CSC then the message is ignored, as the client has already rolled back. Otherwise, the client must update their own CSC to match SI and rollback (line 4). The rollback depends on the type of Response message: if the Response message is due to missed message request then the client should rollback to just after the last successful action using its execution log (line 6). If the Response message is due to irreconcilable message request, then the client local data set is rolled back to the state seen in the AC in the Response message using its execution log and the updated state and logical clock values received from the server are applied to the client's local set (lines 8-10).

Once this is completed, or there were no messages received from the server, the client can send a new message. The client action clock is incremented by one (line 20) and a Result message is created and sent to the server (line 21); the Result

Algorithm 1 Client side algorithm

```
Client $c$;                          /* the client*/
Integer $c_{CSC}$;                   /* client session clock */
Integer $c_{CAC}$;                   /* client action clock */
Shared State $DSc$;                  /* client local replica */
list of Message $messages$;          /* messages received from s */
list of ack;                         /* periodically acknowledgement received from s */

1: if $messages \neq \phi$ then
2:    for each Message $m$ in $messages$ do
3:       if $m_{SI} > c_{csc}$ then
4:          $c_{csc} := m_{SI}$;
5:          if $m_{type} = MMR$ then
6:             rollback $DSc$ to $m_{AC}$;
7:          end if
8:          if $m_{type} = ARMorEARM$ then
9:             rollback $DSc$ to $m_{AC}$;
10:            apply $m_{updates}$ to $DSc$;
11:         end if
12:      end if
13:   end for
14: else
15:   for each Acknowledgement in $ack$ do
16:      if $m_{AC} > C_{CAC}$ then
17:         remove $m_{CAC}$ from $log$;
18:      end if
19:   end for
20:   $c_{CAC} := c_{CAC} + 1$;
21:   send $[UMR, c, di, CDI, c_{CSC}, c_{CAC}]$ to $s$;
22: end if
```

message contains the data item state, the CDI of the data item, the CSC, and the CAC. However, in order to avoid the unbounded growth of the clients logs, the client periodically removes successful actions they submit (line 15-19).

3.5.2 Server Side Algorithm

The baseline server algorithm given in Algorithm 2 executes to processe arrived messages from clients. When there are messages to process, the first is removed and the session identifier, the action clock and logical clock value checks are performed. The session identifier check is made first (line 4); if the server's session identifier (SI) is not equal to the received session identifier (CSC), then this message is ignored (line 18).

If the message passes the session identifier check then the server now compares the action clock, if it is greater then a reqired message from a client is missing. The server incerments a client's session identifier by one and send a missed message request to the client (lines 5-7). The logical clock ensures that client used the same state of the data item as the one held at the server. If the logical clock values are not equal then the client used an out of date version and this message is considered a conflict (line 9). In the event of a conflict, the server advances the SI of the client by one and create authoritative rollback message. The message contains the latest LC of the data item the action attempted to access, the value of the data item, the AC and the SI value (this is the irreconcilable message request mentioned in subsection 3.2.2). Then the server sends the authoritative rollback messagea to a client.

If the clients message passes these two checks then the state and logical clock are updated to reflect the clients action (line 14).

The server periodically sends an acknowledgement message to inform clients of their successful action (line 15).

Algorithm 2 Server side algorithm without backoff

```
Client $c$;                              /* the client*/
Server $s$;                              /* the server */
Shared State $DSs$                       /* shared state */
list of Message $messages$;              /* messages received from clients */
list of Integer $AC$;                    /* action clock for each client */
list of Integer $SI$;                    /* session clock for each client */
list of Message $rollbacks$;             /* rollback message for each client */

1:  while true do
2:    if $messages \neq \phi$ then
3:      Message $m$ := first message in $message$;
4:      if $m_{CSC} = SI[m_c]$ then
5:        if $m_{CAC} >= (AC[m_c] + 2)$ then
6:          $SI[m_c] += 1$;
7:          send $[MMR, AC[m_c], SI[m_c]]$ to $m_c$;
8:          elseif $m_{CAC} = AC[m_c]$ then
9:          if $LC$ for $m_{di}$ in $DS_s > m_{CDI}$ then
10:           $SI[m_c] += 1$;
11:           send $[ARM, AC[m_c], SI[m_c], updates]$;
12:         end if
13:       else
14:         update $m_{di}$ in $DS_s$ with $m_{di}, m_{CDI} + 1$;
15:         send $[ack, AC[m_c]]$;
16:       end if
17:     else
18:       skip;
19:     end if
20:   end if
21: end while
```

3.6 Contention Management

We now explain how the framework described thus far can be augmented with contention management to achieve greater performance (i.e. fewer irreconcilable differences providing less rollback without hindering overall throughput). To achieve this, we exploit the causality inherent in the overall system. As we aim to satisfy application types that exhibit strong causality requirements, we believe this approach as appropriate.

As with all contention management schemes, we exploit a degree of predictability to achieve improved performance. We assume that causality across actions is reflected in the order in which a client accesses shared data items. The diagram in figure 3.2 describes this assumption.

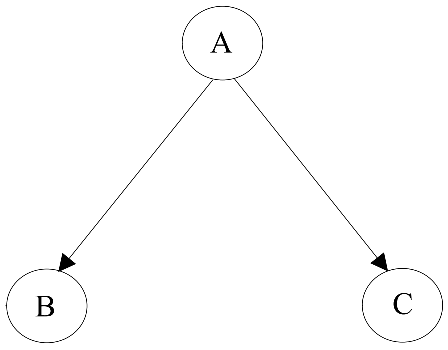

Figure 3.2: Relating action progression to data items

Considering the example shown in figure 3.2, we now describe the essence of causality exploitation in our contention management scheme. We show three data items (A, B, C). If a client, say $C1$, has carried out an action that successfully accessed data item A, we can state the following: there is a higher than average chance that data item B will be the focus of the next action carried out by $C1$; there is a higher than average chance that data item C will be the focus of the next action carried out by $C1$; there is a chance (less than going to B or C) that another data item (which could be anywhere in the shared state) will be the focus of the next action carried out by $C1$.

The server maintains shared data as a directed graph, as mentioned in subsection 2.1.4. The graph created with probabilistic edges indicates the likelihood of clients accessing individual items of shared state given the last item of shared state they accessed. The graph is defined as G=(V, A, VV) where $V(G)$ is s set of vertices representing the data items in the server's data set, $A(G)$ is a set of ordered pairs of vertices identifying the edge between two data items and $VV : V(G) \rightarrow X$ where X is the volatility for a given vertex ($X \geq 0$). The client is not aware of the graph during execution. To achieve this we add a number of additional constructs at the server side:

- Volatility value (VV): a value associated with each data item indicating its popularity. Whenever a successful action accesses a data value, its volatility value is incremented by 2 and the neighbouring data items' volatility values are incremented by 1.

- Delta queue (DQ): those actions that could not be honoured by the server due to out of date LC values are stored for a length of time on the DQ. This length of time is calculated as the sum of the volatility value of the data item where the out of date LC was discovered, together with the highest volatility values of data items up to three hops away on the graph. We call this the DQ time of a client.

- Enhanced authoritative rollback message ($EARM$): when an action request is de-queued from the DQ, an enhanced authoritative message is sent to the client who initially sent the action request. An enhanced authoritative rollback message extends the authoritative rollback message with all shared states and the LC values of up to three hops away in the graph of the most volatile nodes.

We now use an example to clearly describe how our backoff contention management works. In figure 3.3, we show a graph created with probabilistic edges held by the server with volatility values shown next to nodes. The length of both the predication list and calculating the backoff time is set to three in this example. A graphical representation G=(V, A, VV); for figure 3.3 the following set exists:

$$V(G)=\{A, B, C, D, E, F\}$$

$$A(G)=\{(A, B), (A, C), (C, B), (B, D), (C, E), (C, F), (F, E)\}$$

$$VV(A)=50,\ VV(B)=5,\ VV(C)=10,\ VV(D)=7,\ VV(E)=25,\ VV(F)=40$$

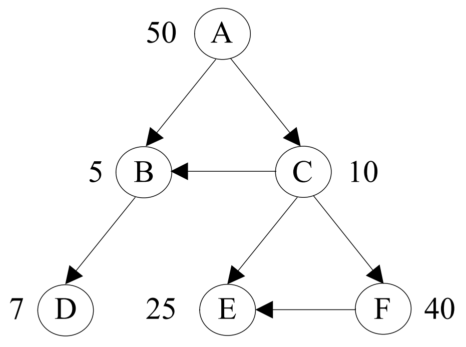

Figure 3.3: Graph with volatility values

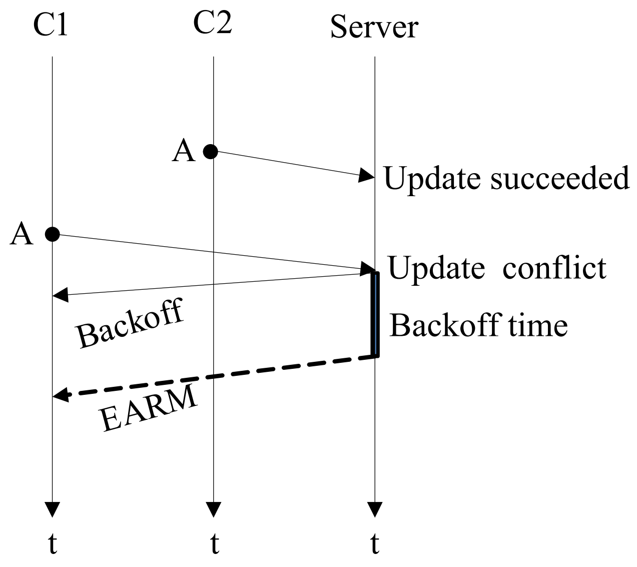

Figure 3.4: Message passing between clients and server

Assume the server receives a client message from $C1$ indicating access to shared data item A, as shown in figure 3.4. Unfortunately, the request was irreconcilable due to out of date LC values. Since the successful action of $C2$, the logical clock for A has been updated prior to receiving $C1$'s message.

The action request from $C1$ is placed on the delta queue. The DQ time for $C1$ is the summation of the volatility values of A, C, F, and E ($50 + 10 + 40 + 25 = 125$). These are the highest volatility values up to three hops away. Therefore, each time the client carries out an action, all the DQ time values are decreased by one (see pseudo-code in Algorithm 3). For $C1$, the server will have to loop 125 time unit before its action request can be de-queued. Once de-queued the enhanced authoritative rollback message is sent to $C1$, complete with all the data item values and LC descriptors of A, C, F, and E (the LC values are not shown on the diagram). On receiving the enhanced authoritative rollback message, $C1$ can update A, C, F, and E in its own local replica and rollback, as instructed.

The approach we have taken is mostly a server side enhancement. We took this design decision to alleviate clients from the burden of participating in contention management. The only enhancement at the client side is the ability to update an additional number of data items in its own replica of shared state on receiving a rollback message.

If the contention management approach was left to run as described, then the volatility values would continue to rise and action requests would spend ever longer on the DQ. To prevent this, we take a very simple approach in that, we zero volatility numbers when they reach 200.

There are quite a number of parameter values that we have described that can be changed (e.g. increasing volatility by 2 and neighbours by 1, determining DQ time, and assigning 200 as the ceiling value for DQ time). This was done to add clarity to the descriptions and these are the values we use in our evaluation. These values were inspired from backoff schemes, specifically the BEB and PLEB algorithms (see section 2.8.1). We found that these values provided a suitable environment that exhibits the benefits of our contention manager in this particular style of application under these throughput conditions. However, an application developer may experiment with other values.

3.6.1 Server Side Algorithm with Contention Management

The server algorithm given in Algorithm 3 executes in a loop, either processing arrived messages or maintaining the delta queue. When there are messages to process, the first is removed and the state of the client checks is performed. The state of the client can be in one of two states:

- $Progress$: last client message could be honoured
- $Stalled$: last client message could not be honoured or was ignored

If the client is in the progress state then the CAC is checked first. If it is two or greater than the AC of the client held by the server, then the server increments the client's SI and AC by one and moves the client to a stalled state (lines 4-8). Otherwise, the logical clock of the data item checks is performed. The logical clock ensures that the client uses the same state of the data item as the one held at the server. If the clock values are not equal, then the client uses an out of date version and this message is considered an irreconcilable action (line 10). In this case, the session identifier at the server is incremented by one and the client moves to the stalled state (lines 11-12). This means that any more messages sent by the client in the same session will not be processed. The resulting message is put in the delta queue (lines 13-14) for the required time period, based on the volatility values seen in the graph. The time period is calculated by taking the volatility value of the highest volatile neighbour of the data item that conflicted. From that vertex, the highest volatile neighbour is chosen. The sum of the resulting volatility values then forms the backoff time. The number of vertices explored is a simulation parameter that can be configured as required.

If the client is in the stalled state, the CAC is checked against the AC of the client held by the server. If it is greater, the server increments the client's session identifier by one and an MMR message is sent to the client containing the action clock of the client held by the server (lines 20-24). This means a required message from the client is missing. If the CAC is equal, the AC of the client held by the server and client uses the same state of the data item as the one held at the server; the client's state is moved to the progress state, and then the state and logical clock are updated to reflect the client's action (lines 32-34). Otherwise, the message is put in the delta queue for the required time period based on the volatility values seen in the graph (lines 26-29).

Algorithm 3 Server side algorithm with backoff

```
Client $c$;                          /* the client*/
Server $s$;                          /* the server */
Shared State $DSs$                   /* shared state */
list of Message $messages$;          /* messages received from clients */
list of Integer $AC$;                /* action clock for each client */
list of Integer $SI$;                /* session clock for each client */
list of Status $status$;             /* status for each client */
 1: while true do
 2:   if $messages \neq \phi$ then
 3:     Message $m$ := first message in $message$;
 4:     if $status[m_c] = progress$ then
 5:       if $m_{CAC} >= (AC[m_c] + 2)$ then
 6:         $SI[m_c] += 1$;
 7:         $AC[m_c] += 1$;
 8:         $status[m_c] = stalled$;
 9:         send $[MMR, Ac[m_c], SI[m_c]]$ to $m_c$;
10:         elseif $LC$ for $m_{di}$ in $DS_s > m_{CDI}$ then
11:         $SI[m_c] += 1$;
12:         $status[m_c] = stalled$;
13:         generate $DQ\ time$ for $m_{di}$;
14:         add $m$ to $delta$ for $DQ\ time$;
15:       else
16:         update $m_{di}$ in $DS_s$ with $m_{di}, m_{CDI}$;
17:         updateVolatility($m_{di}$, $VV$, 0);
18:         send $[ack, AC[m_c]]$;
19:       end if
20:       elseif $status[m_c] = stalled$ then
21:       if $m_{CSC} = SI[m_c]$ then
22:         if $m_{CAC} > AC[m_c]$ then
23:           $SI[m_c] += 1$;
24:           send $[MMR, AC[m_c], SI[m_c]]$ to $m_c$;
25:           elseif $m_{CAC} = AC[m_c]$ then
26:           if $LC$ for $m_{di}$ in $DS_s > m_{CDI}$ then
27:             $SI[m_c] += 1$;
28:             generate $DQ\ time$ for $m_{di}$;
29:             add $m$ to $delta$ for $DQ\ time$;
30:           end if
31:         else
32:           $status[m_c] := progress$;
33:           remove $rollbacks[m_c]$;
34:           update $m_{di}$ in $DS_s$ with $m_{di}, m_{CDI} + 1$;
35:           updateVolatility($m_{di}$, $VV$, 0);
36:           send $[ack, AC[m_c]]$;
37:         end if
38:       else
39:         send $rollbacks[m_c]$ to $m_c$;
40:       end if
41:     end if
42:   end if
43:   updateDetlaQueue($m$);
44: end while
```

At this point (line 35), the volatility of the graph needs to be updated to reflect the use of this data item (see Algorithm 4). The data item that the client used is directly updated along with all the neighbour data items, according to the graph. The amount by which the volatility is increased is a simulation parameter, but it would be expected that the data item that was used directly will have its volatility increased by a larger value than the neighbouring data items.

Algorithm 4 volatility update algorithm

```
**function** updatevolatility($m_{di}$, $VV$, $Threshold$)
**Graph** $G$;
**Data Item** $di$;
**Integer** $P$;            /* Parameter */
**Integer** $VV$;           /* Volatility Value*/

 1: for all $(x, y) \in A(G)$ do
 2:   if $Threshold < 3$ then
 3:     if $x := m_{di}$ then
 4:       $VV(x) := VV + (P + 1)$;
 5:     else
 6:       $VV(y) := VV + P$;
 7:     end if
 8:     $Threshold := Threshold + 1$;
 9:   end if
10: end for
```

For every iteration of the loop, excluding when a message conflicts (line 43), the delta queue is updated (see Algorithm 5). The update checks every message in the delta queue and reduces the time value by one. After updating the time, if the value is zero then this message has expired and a Response $EARM$ message is created. This message contains the updated state and logical clock for the data item used, as well as the action clock and session identifier of the client held by the server, and the data items that the server predicts the client is likely to use.

Algorithm 5 Delta queue algorithm

```
function updateDeltaQueue($m$)
list of Message $delta$;        /* delta queue */

1: for each Message $dqm$ in $delta$ do
2:     if $DQ\ time$ for $dqm > 0$ then
3:         set $DQ\ time$ for $dqm - 1$;
4:     end if
5:     if $DQ\ time$ for $dqm = 0$ then
6:         generate $updates$ for $dqm_{di}$;
7:         send $[EARM, AC[m_c], updates]$;
8:         remove $dqm$ from $delta$;
9:     end if
10: end for
```

From the original conflicting data item, an access prediction is created for the client. This is based on a trace through the graph, where the highest volatile neighbour is found and added to the prediction list. This process continues, with the highest volatile neighbour becoming the next vertex to investigate in the next iteration. The length of the trace is a simulation parameter that can be set as appropriate. Once the message has been created, this is sent to the client and the old message is removed from the delta queue.

3.7 Framework Properties

The framework described thus far can be rationalised in the following manner:

- Liveness: clients progress until a server informs them they must rollback (via authoritative rollback message). If this message is lost in transit, the client will continue execution anyway, sending access notification messages to the server. The server will keep responding to these messages with the authoritative rollback message until the client finally replies appropriately. If the client message that is a direct response to the authoritative rollback message goes missing the server will eventually realise this due to receiving client messages with the appropriate SI but CAC values that are too high. This will cause the server to respond with a new authoritative rollback message.

- Causality: a client always rolls back to where an irreconcilable action (or missing action due to message loss) was discovered by the server (liveness). Therefore, all actions that are reconciled at the server and removed from a client's execution log maintain causality. Those actions in the execution log are in a state of reconciliation and may be rolled back.

- Eventually consistent: if a client never receives a message from a server then either: **(i)** all client requests are honoured and states are mutually consistent or **(ii)** all server or client messages are lost. Therefore, as long as sufficient connectivity between client and server exists, shared data will become eventually consistent.

The framework described provides an opportunity for clients to progress independently of the server in the presence of no message loss and no irreconcilable issues on the shared data. However, the burden of rolling back is much more substantial than other eventually consistent optimistic approaches (e.g. Bayou and IceCube), as subsequent actions that occur at a client after an irreconcilable or message loss event are void. This does, on the other hand, provide the benefit of not requiring any application level dependencies in the protocol itself, the application developer does not need to specify any exception handling facility to satisfy rollback. Actions that are deleted from a client's execution log are considered stable.

3.8 Summary

In this chapter, we presented our framework architecture and its requirements. Then, we described the protocol design which enables an optimistic replicated system to reduce the number of update conflicts. We also stated the protocol's assumptions and environments. The protocol was illustrated by pseudo-code. Finally, we demonstrated how the framework was augmented with contention management to handle the updated conflicts and preserve causal relations between client actions by giving a generic example.

In the next chapters, we focus on the practicality of our framework. We start the next chapter by describing the simulation design for measuring the performance of our framework.

Chapter 4

Framework Implementation

The preceding chapter provided a formal description of our framework. This allows the framework to be theoretically understood and its function formally presented. In this chapter, we will describe the implementation of the framework using simulation. We chose simulation since a simulated environment has the advantage of allowing us to vary the system parameters without physically changing the software or hardware. Furthermore, simulation allows us to concentrate more on the behaviour of the framework than the complexity of the implementation details, such as the network environment and the message passing between clients and server.
This chapter describes the simulation system, which will be the basis for the results presented in the next chapter.

4.1 Simulation Architecture

The simulation system was implemented using Java programming language and Windows operating system. The architecture of the simulation was composed of a single server process, a number of client processes and a messaging service. The following subsections give a more detailed description of the architecture.

4.1.1 Clients

The simulator runs a number of client processes, implemented by class Client. The client class is associated with a number of different classes that generate a workload for each client. A workload generates a number of logical clocks to aid clients' management of their execution. The client class also includes rollback mechanism to restore the local copy to a previous consistent state in the case that clients' actions have not been honoured by the server.

4.1.2 Server

The server process that frequently accepts messages from clients was implemented by class Server. The server class maintains data items on behalf of clients and is responsible for generating a number of logical clocks for each data item and for each client. The server class is also responsible for handling irreconcilable actions between different clients. The other entities associated with the server class are:

- Informing clients when to rollback
- Calculating the backoff time of unsuccessful messages
- Predicting data items to clients that are likely to be used in the future
- Preserving the causal relationship between a client's actions, and
- Ensuring that clients' replicas are eventually consistent.

4.1.3 Messaging Service

The simulator runs HornetQ [64] as its messaging service to manage the communication between clients and the server and provide an asynchronous message delivery layer. HornetQ is a high performance asynchronous messaging system from JBoss. It provides a message queue pattern. In this service, clients communicate with a server in an indirect manner. A client sends a message to a queue, and then some time later HornetQ delivers the message to the server.

4.2 Simulation Environment and Setting

The simulation environment was executed on a single computer. The computer has a 3 GHz Intel Dual core CPU, with 4 GB of RAM. The operating system was Windows 7. Communication delays were introduced for the client process to mimic those found in a real world system. Before sending a message, a client process was paused for a random time ranging between 100 and 1000 milliseconds to create a communication delay.

The graph was created with probabilistic edges to aid in informing the contention manager used for the simulation located at the server side. It was generated randomly with an ability to determine the maximum number of edges leading from a vertex. Vertices within the graph could have no edges leading from them, whereas others are permitted the maximum numbers of edges. Edges cannot loop back to their starting vertex (they have to point to another node in the graph). We chose a maximum of three edges for all the experiments.

With the contention management, when updating the volatility, the vertex (data item) that the client directly touched is increased, alongside the neighbouring ones (vertices that are linked by outgoing edges from the original vertex). The value of the volatility is a simulation parameter. In this simulation, to prevent the volatility values from continuing to increase and action requests from spending a longer time in the delta queue, we zero the volatility numbers when they reach a specified ceiling. The values were used in the simulation are inspired from backoff schemes, specifically the BEB and PLEB algorithms (see section 2.8.1). We found that these values provided a suitable environment that exhibits the benefits of our contention manager in this particular style of application under these throughput conditions.

The length of the trace, when generating the access prediction, is also a simulation parameter. This means that the prediction list will contain more than one data item or a minimum of one data item (the original conflicting data item). The reason the prediction may only contain the original item is if a vertex in the graph has no outgoing edges. As such, the assumption is that the client's next operation would be for an unrelated data item.

The actions each client will make are generated before each test and are based upon the structure of the graph created with probabilistic edges. From a random data item, an action is created involving that data item. A random outgoing edge of the data item is chosen to form the next action. This process continues either until the number of actions required has been reached or a vertex has been chosen that has no edges. If the vertex has no edges, then a new random data item is chosen.

The rollback mechanism implemented was to cache the data set before sending the message. When an action conflicts, the cached data set simply replaces the current data set and the updates are then applied. To achieve this, each message has to have an identifier to link the message sent with the cached copy of the data set. When a cache data set is no longer required (the message for which it was cached was successful), it is removed to reduce the storage overhead. The rationale for choosing to cache the data set for the rollback mechanism was to simplify the client side algorithm. Each action is recorded forming a historical list of actions. When a message is received from the server indicating a conflict, then the messages in the history could be used to reverse the change to the data item that the action performed. In this scheme, the successful actions in the history can be removed safely, while any message that came after the conflict message will have to be reverted. This scheme does not require the whole data set to be cached, only the actions that potentially need to be rolled back.

4.3 Summary

In this chapter, we have described the simulation design and the simulation setting. The description specified the main components of the simulation package and outlined its architecture. Also, we introduced the simulation running environment and explained the mechanism of the simulation.

The next chapter presents a series of experiments based on the above design to analyse the performance of the protocol.

Chapter 5

Framework Evaluation

In order to evaluate the performance of our framework, we conducted a series of experiments. The experiments carried out in the simulation described in the previous chapter. Therefore, this chapter presents a variety of experiments followed by the results obtained of such experiments for justifying the validity of our work.

To compare our contention management, as detailed in chapter 3, two versions were implemented:

1. The baseline protocol as presented in algorithm 2. This protocol does not make use of the delta queue to backoff a client process. The client operates in the same way as in the contention management protocol, but if there is a logical clock conflict at the server, a message will be sent to the client straightaway. This message contains the updated state and all values for the data item that the client conflicted against but not the server's prediction for the client.

2. The protocol with contention management as presented in algorithm 3.

This comparison will allow us to make conclusions as to which protocol should be used.

In both protocols, we investigate their performance with various parameters in order to demonstrate the irreconcilable actions rate at the server, as well as throughput. However, some of the parameters are constant since their variation does not change in the behaviour of the protocols. The simulation parameters are listed in table 5.1.

Parameter	Value
Number of clients	2-100
Pause time between client's actions	100-1000 ms
Number of data items	100-1000
Maximum number of edges per data item	3
Length of predication list	3
Increasing volatility of target data item	2
Increasing volatility of neighbours data item	1
Volatility ceiling	200

Table 5.1: Simulation parameters for protocols evaluation

The experiments were carried out to measure the performance of both protocols in terms of reducing the number of update conflicts. Reducing the number of update conflicts is usually considered a major indicator of superiority for reconciliation protocols (see section 2.4). We also measured the system throughput under different sizes of data items and different numbers of clients. The throughput is measured as the number of successful committed actions over a given interval of time at the server.

5.1 Experiments

This section describes a series of experiments. The first set of experiments determines if backoff contention management can reduce irreconcilable actions. The second set of experiments determines if throughput would be adversely affected by the introduction of backoff contention management. The final set of experiments determines if the server's execution time would be affected by including backoff contention management. For each set, we created many graphs with varying numbers of data items (100, 300, 500 and 1000). For each graph, we ran the experiments with a different number of clients (2, 5, 10, 25, 50 and 100). Each experiment was run a number of times to gain an average figure for analysis.

5.1.1 Experiment 1: Irreconcilable client actions (conflicts)

This experiment provides a first performance analysis in order to provide a feel for whether the effect of our protocol reduces the number of update conflicts. The number of conflicts is measured by counting the total number of irreconcilable actions at the server for a particular number of clients. In this experiment, we have set different numbers of data items with different numbers of clients. The four graphs presented in Figure 5.1 show the number of conflicts for different client sizes. Each graph displays the number of conflicts recorded for each different graph size.

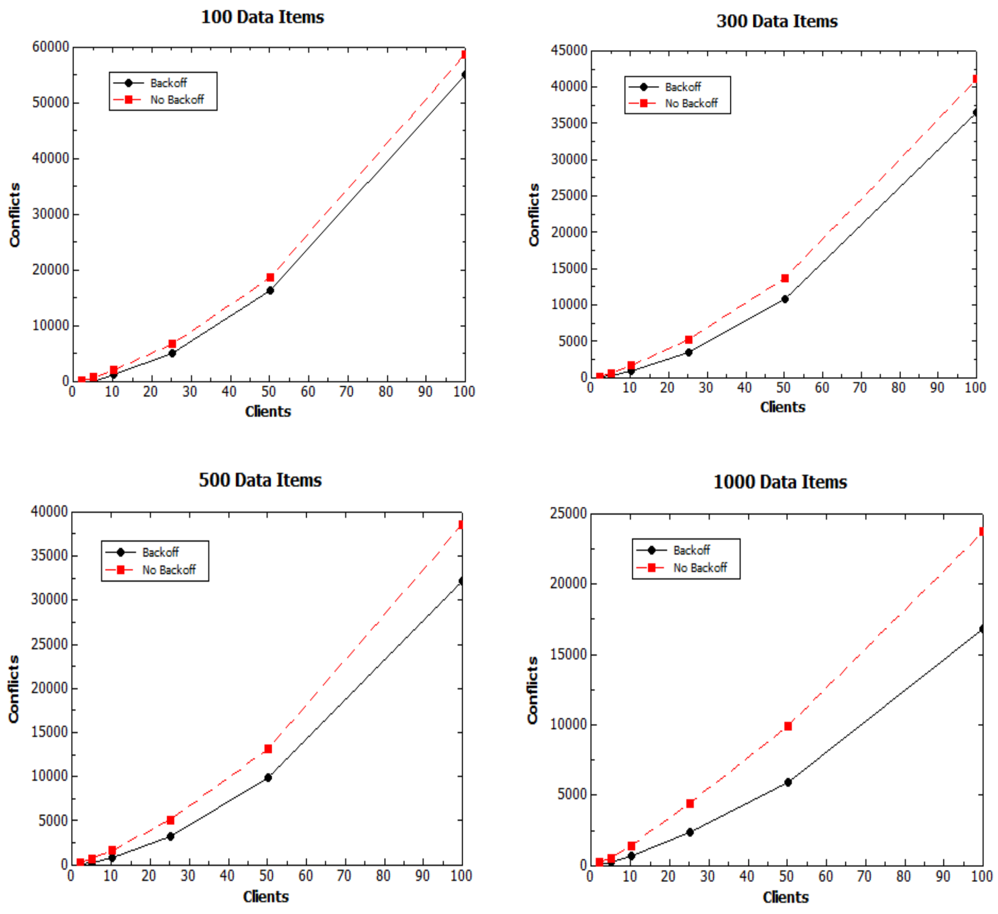

Figure 5.1: Irreconcilable actions with varying graph size

The first observation to be made is that including backoff contention management lowers conflicts in all graph sizes and for variable numbers of clients, indicating that fewer client actions were irreconcilable (fewer conflicts). As the graph becomes larger, both protocols result in fewer conflicts. This indicates that the larger the replica state, the less chance there is of irreconcilable differences occurring (which is quite obvious). However, the introduction of backoff contention management results in fewer conflicts for all client numbers in all graph sizes.

The improvement gained by including backoff contention management shows itself more significantly as graph sizes increase. This indicates that backoff contention management provides a significant improvement in environments where conflicts do occur, but proportionally less as the size of the replica state increases: if we consider 100 clients, the most favourable circumstance (1000 data items) shows an improvement of approximately 28%, compared to an approximate improvement of 6% in the least favourable circumstance (100 data items).

5.1.2 Experiment 2: Throughput of successful client actions

The second set of experiments measured the throughput at the number of successful committed actions at the server. In this set, we conducted two types of experiments. The first experiment measured the number of actions committed throughout the entire simulation. In this experiment, we also have a set different numbers of data items with different numbers of clients. The graphs in figure 5.2 show the throughput in commits per second on the vertical axis versus the varying number of clients on the horizontal axis.

The first observation to be made is that including backoff contention management improves throughput for all graph sizes and all ranges of client numbers. We can see in the graphs that when the number of data items increases, the throughput increases. This is to be expected, as with a larger graph conflicts against the same data item will be more infrequent. Again, if we consider 100 clients, the most favourable circumstance (1000 data items) shows an improvement of approximately 29%, com-

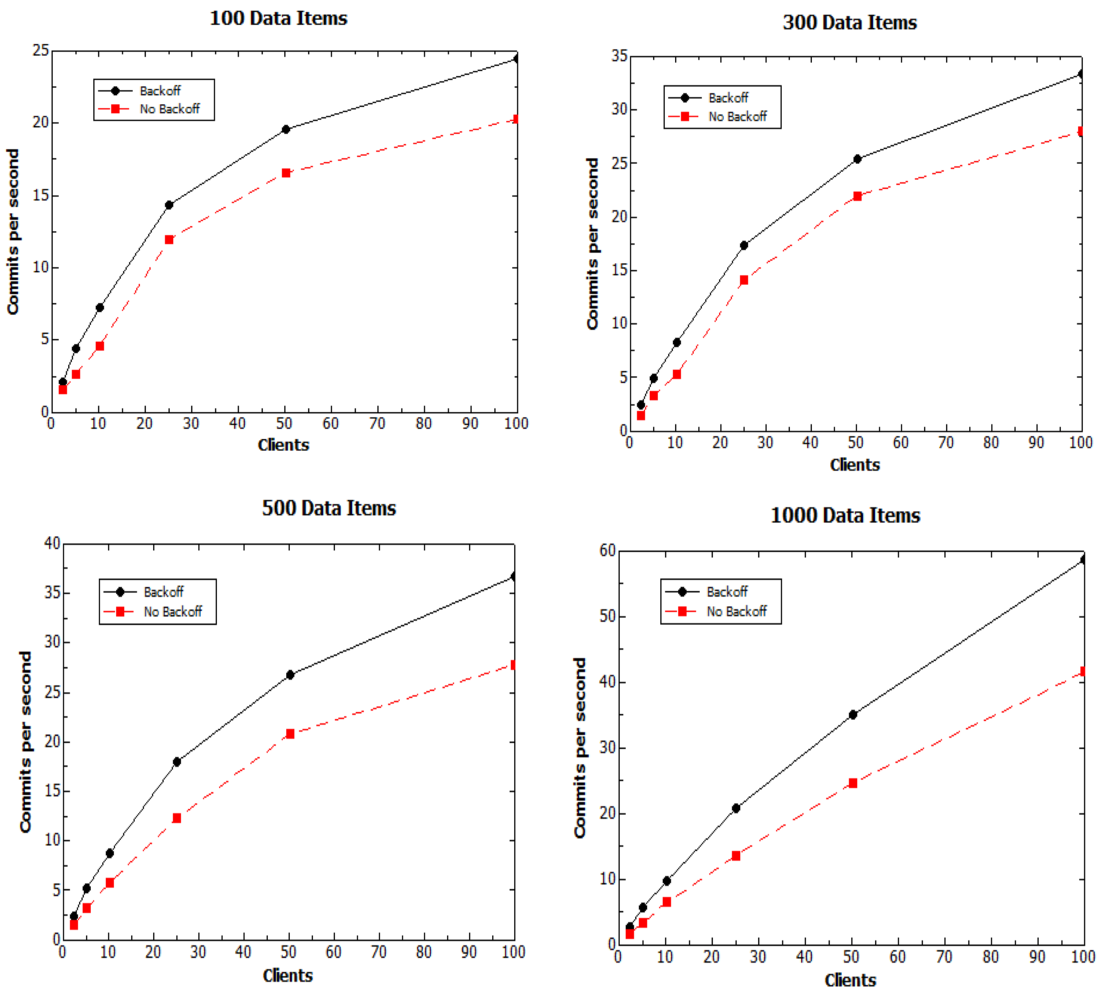

Figure 5.2: Throughput with varying graph size and increasing number of clients

pared to an approximate improvement of 16% in the least favourable circumstance (100 data items).

The second experiment measured the number of committed actions every five seconds. Figure 5.3 shows a graph size of 100, 300, 500, and 1000 data items, with a varying client size of 25, 50, and 100. Each client performs two hundred update actions, and then terminates.

The first point to note is the initial large number of successful actions for both protocols in all graph sizes. This is reasonable because there is no high contention workload on data items at the beginning and therefore the conflicts are rare. However, it can be seen in each graph that the number of successful actions of backoff protocol is significantly higher than that for no backoff protocol when the system

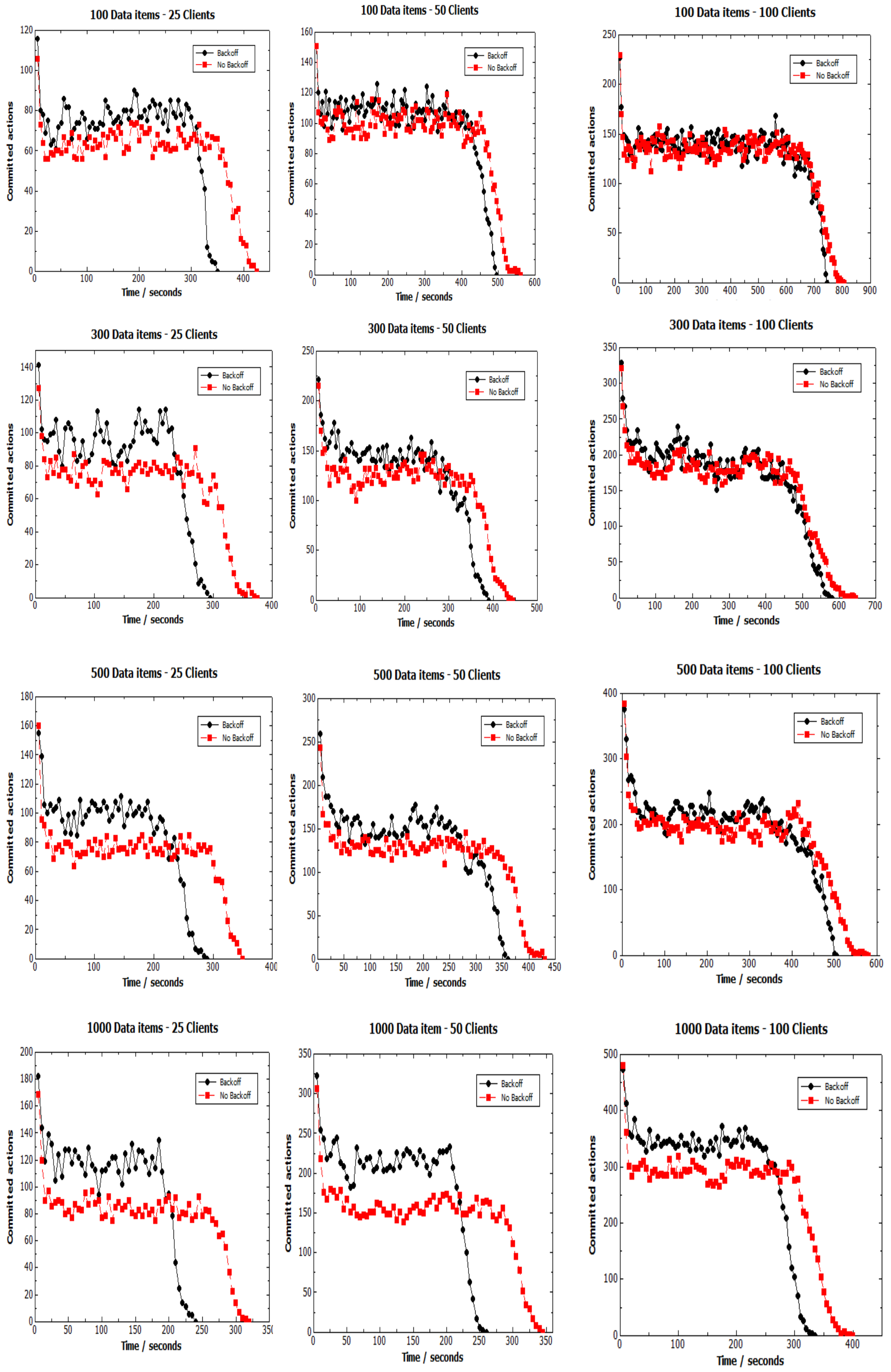

Figure 5.3: Throughput with varying graph and client size

stabilizes. This is due to the fact that the backoff protocol guarantees a lower conflict rate than the no backoff protocol under high data contention workload, as shown in figure 5.1.

The drop at the end for each protocol indicates that the number of clients active in the system reduces as they finish and leave the system. Another observation for all graphs is that the time taken to process clients' messages by a server is less with backoff protocol. This is to be expected since backoff contention management provides high throughput and therefore clients quickly finish and leave the system.

5.1.3 Experiment 3: Execution time

This experiment measures the total execution time at the server. We define the execution time as the time elapses from when the server starts executing the first action to when the last action completes execution. Therefore, the four graphs in figures 5.4 show the total execution time was spent at the server to handle clients' actions with and without backoff contention management. The execution time is measured for different graph sizes with increasing client numbers.

The first observation to consider is that as the size of the graph increases, the execution time reduces for both protocols. Furthermore, the execution time grows as the number of clients increase. However, the execution time with backoff protocol is less than without backoff protocol, for example if we consider 100 clients on 100, 300, 500, and 1000 data items, it is approximately 6.8%, 10%, 12.9%, and 16% respectively.

Another key point to make here is that increasing the number of data items has a considerable impact on the execution time for both protocols. We can see in all graphs that, as the size of the graph increases, the execution time reduces, with the reason being that the conflicts will be fewer over a greater range of data items. However, including backoff protocol, the execution time always stays lower in all graph sizes even when the number of clients increases.

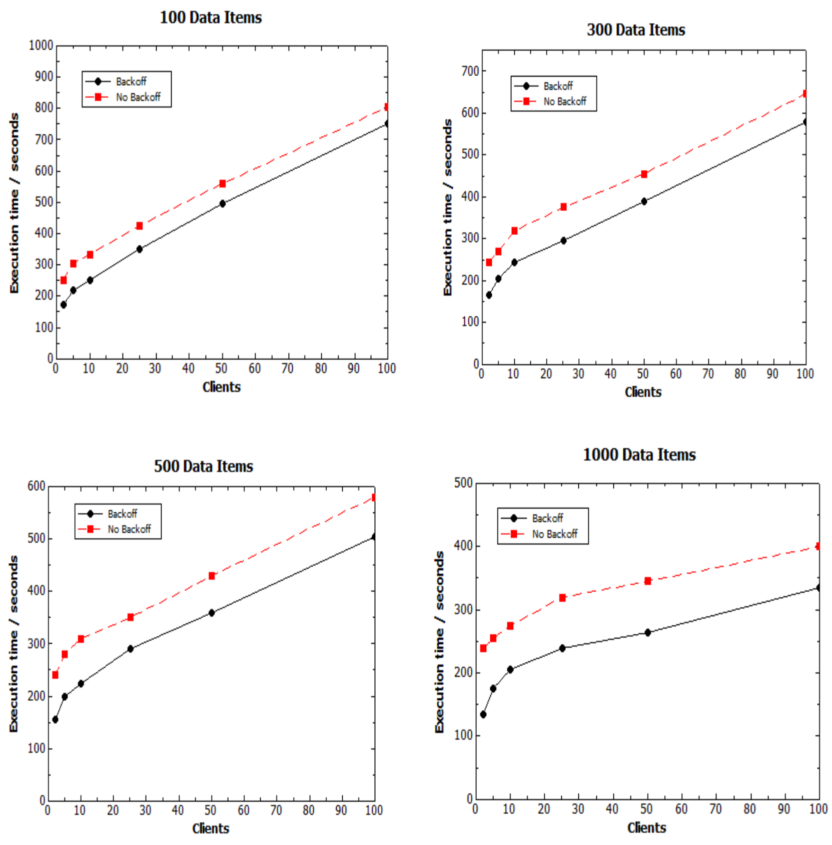

Figure 5.4: Execution time with varying graph size

5.1.4 Experiment 4: Protocol Overhead

We now determine the overhead imposed by our protocol over our benchmark system that simply affords rollback. Our benchmark is the same as in our previous experiments that demonstrated that our contention manager provides increased throughput and lower inconsistency rates. That is, in the benchmark the server simply informs the client that they must rollback and affords a message containing only the most up to date value of the item resulting in inconsistency.

The graphs in figure 5.5 show that our contention manager imposes significant overhead in terms of increased data (message size). This is primarily to do with the need to send multiple data values from the server to the client when inconstancy is realised at the server. This cost rises as a percentage of the overall messages in the system as client numbers rise (contention rises). A conclusion to be drawn is that although our approach significantly improves overall throughput by lowering inconsistency, it does so at the expense of resource requirements (network load and client/server message handling). Therefore, systems employing our approach would be required to consider the support of lower client numbers given an equivalent resource.

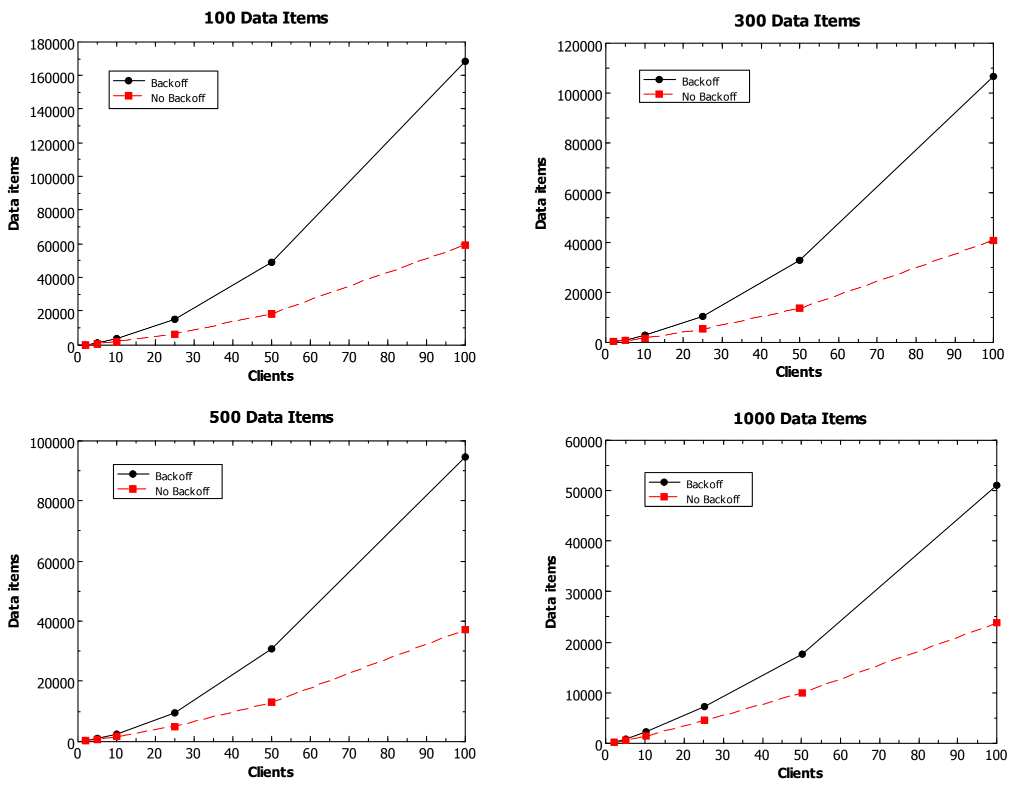

Figure 5.5: Data size with varying graph and client size

5.2 Summary

In this chapter, we measured the performance characteristics of our framework via a series of experiments. These experiments were conducted to justify our framework. The results obtained were based on the two versions: **(1)**backoff contention management, and **(2)** no backoff contention management. The analysis of the performance shows the following:

- The backoff contention management protocol always out performs the no backoff contention management protocol for all graph sizes. This means that introducing backoff contention management can reduce the update conflicts within an optimistic replication scheme.

- In both protocols, as the number of data items increases, the number of conflicts reduces. This is to be expected in both protocols as with a larger number of data items, clients are less likely to access the same data item as frequently as they would when there are a smaller number of data items.

- The measurements indicated that our backoff contention management scheme improved the overall system throughput for all ranges of data contention workload. This scheme has also shown improvement of overall performance for those applications with strong causality requirements when augmented with our protocol.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis has investigated the main issues concerning optimistic shared data access in which domain semantics knowledge can be exploited in the design and implementation of consistency models and reconciliation protocols, to ensure eventual consistency and reduce update conflicts. The thesis pointed out how such semantic knowledge can be used to ease the development of the distributed applications that are required for optimistic replication. We observed that increased instances of application levels results in protocol overheads.

The thesis analysed the advantages and limitations of existing approaches in optimistic replication, semantic reconciliation and contention management policies, and therefore the thesis has focused on building a distributed application which will ease the design burden from application programmers where the causality requirement is too high within the client's actions.

The thesis has described a framework within which contention management can be provided by exploiting semantic relations between data items. We used client/server architecture as an implementation basis for our framework. A server was enhanced to enforce the backoff based contention management scheme. A client plays no part in determining backoff, allowing the backoff scheme to appear transparent to client implementations. This also makes the contention management scheme independent from whatever shared data access protocol is used to enforce the required consistency model.

Our evaluation, via experimentation, demonstrates how the backoff contention management scheme improves overall performance by reducing irreconcilable actions on the shared state while increasing throughput. A number of experiments were carried out and the backoff contention management improved performances in all scenarios (e.g. heavy to light congestion).

We acknowledge that our approach will only perform better than existing optimistic approaches if causality is a critical factor. That is, it will perform better for those applications within which all existing client actions occurring after the rollback checkpoint are determined void, when resolving irreconcilable differences in the shared state. However, we believe that such applications can benefit from optimistic replication if used together with a backoff contention management scheme that utilises causality in its prediction of client actions.

To the best of my knowledge, this is the first time optimistic replication has been combined with causality informed backoff based contention management. We have shown this combination to benefit applications with strict causality requirements. As such, we believe that this is not only a useful contribution to the literature, but opens new avenues of research by bringing the notion of contention management through client backoff to the attention of shared data based on replication schemes.

6.2 Future work

This section contains a discussion of paths for potential future research. We will here list what we think are interesting directions to develop our work.

6.2.1 Decentralization

Although our approach is significant because it provides the benefit of not requiring any application level dependencies in the protocol itself, we cannot use the framework in a peer-to-peer system because its contention management scheme and the shared state were designed to be maintained by a centralized server. The approach could be extended to peer-to-peer based evaluation and create backoff contention management for mobile environments where epidemic models of communication are favoured.

6.2.2 Dynamic graph

We created the graph with probabilistic edges to aid the process of informing the contention manager, specifically for the purposes of experimentation and associated client behaviour to the graph. The use of the created graph with the probabilistic edges structure accurately mimics the typical relations seen between data items and users' actions. For example, Amazon's website provides a section on each product page showing other products customers have purchased after having bought the product displayed. Here, we see how a graph structure could be created among products listed, with the vertices being the product and the edges forming the semantic link that customers generate over time.

The structure of the created graph with probabilistic edges could be enhanced to change dynamically to reflect client behaviour. Initially, we could consider all data items as being connected to one another directly. Over the lifecycle of the system, the graph with probabilistic edges could be modified to remove edges between items that have never experienced a semantic link. Using this, the created graph with probabilistic edges will change over time and illustrate how clients' actions are made between the different data items.

6.2.3 Mixed-Mode Operation

As mentioned in subsection 6.2.2, the created graph with probabilistic edges could be enhanced to change dynamically to reflect clients' behaviour. As a result of this, relevant information could be extracted to create a causal history graph to allow the framework to provide flexible causality preservation. By flexible causality, we mean that some clients may need stronger causality than others (i.e. some clients need 60% causal ordering, some need 100 %, and some have no at 0%). Therefore, depending on the available information in the causal history graph, the framework can provide a flexible judgement to maintain causality for clients only in the situations in which they are needed.

6.2.4 Flagging Early Conflicts

The notification of a conflict is delayed in back-off. This has the desirable effect (as already mentioned) of not inhibiting the client. However, this may not be desirable in some application types, where there may be benefit in a client being notified early that conflict has occurred (and so allow a client to proactively alter their current execution). This issue is beyond the scope of this thesis and such application types are not considered. However, future work could consider hybrid versions of the approach presented here, where early flagging of conflict is possible while still maintaining the appropriate back-off scheme. It could be that a client could decide, once flagged, to instruct the server to remove the conflicting action from the back-off queue. Alternatively, the client may simply acknowledge the conflict by altering their own execution strategy and data access patterns, possibly carrying out work not involving shared state until the conflict is raised in the normal manner via the back-off approach, at which point the client may proceed as described within the thesis.

References

[1] S. J. Mullender, *Distributed Systems.* ACM-Press, 1993. 1

[2] S. Mullender, *Introduction to distributed systems.* CERN, 1992. 2

[3] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design.* Addison-Wesley Longman, 2001. 2, 24

[4] S. Goel and R. Buyya, "Data replication strategies in wide area distributed systems," ISBN 1-599044181-2, Idea Group Inc., Hershey, PA, USA, Tech. Rep., 2006. 4

[5] S. Son, "Replicated data management in distributed database systems," *ACM SIGMOD Record*, vol. 17, no. 4, pp. 62–69, 1988. 4

[6] S. Rahimi and F. Haug, *Distributed database management systems: A Practical Approach.* Wiley-IEEE Computer Society Pr, 2010. 5

[7] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005. 6, 20, 21, 32

[8] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on.* IEEE, 2000, pp. 464–474. 6, 20

[9] Y. Saito, “Consistency management in optimistic replication algorithms,” *INTERNET, ÄOnlineÜ*, vol. 15, pp. 1–18, 2001. 7, 22

[10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. 7, 14

[11] R. Guerraoui, M. Herlihy, and B. Pochon, “Polymorphic contention management,” *Distributed Computing*, pp. 303–323, 2005. 9, 43

[12] A. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems.* Cambridge Univ Pr, 2008. 13

[13] M. Raynal and A. Schiper, “The causal ordering abstraction and a simple way to implement it,” *Information processing letters*, vol. 39, no. 6, pp. 343–350, 1989. 14

[14] A. Schiper, J. Eggli, and A. Sandoz, “A new algorithm to implement causal ordering,” *Distributed Algorithms*, pp. 219–232, 1989. 14

[15] A. Mostefaoui and O. Theel, “Reduction of timestamp sizes for causal event ordering,” *Relatório técnico*, vol. 1062, 1996. 14

[16] M. Loper, “Causal ordering protocols: A survey,” *Draft) en¡ http://www. cc. gatech. edu/people/home/margaret/papers/CausalSurvey-Draft2. pdf*, 2008. 14

[17] C. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66. 17

[18] F. Mattern, “Virtual time and global states of distributed systems,” *Parallel and Distributed Algorithms*, pp. 215–226, 1989. 17

[19] J. Gross and J. Yellen, *Graph theory and its applications.* CRC press, 2006. 18

[20] P. Spirtes, “Directed cyclic graphical representations of feedback models,” in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence.* Morgan Kaufmann Publishers Inc., 1995, pp. 491–498. 19

[21] S. Yi and H. Shin, “Data replication preserving semantic relationship on a single wireless channel,” in *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on.* IEEE, 2009, pp. 595–600. 20

[22] C. Liu and K. Lin, “Efficient scheduling algorithms for disseminating dependent data in wireless mobile environments,” in *Wireless Networks, Communications and Mobile Computing, 2005 International Conference on*, vol. 1. IEEE, 2005, pp. 375–380. 20

[23] J. Gray, P. Helland, P. O'Neil, and D. Shasha, “The dangers of replication and a solution,” in *ACM SIGMOD Record*, vol. 25, no. 2. ACM, 1996, pp. 173–182. 20

[24] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, “Database replication techniques: A three parameter classification,” in *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on.* IEEE, 2000, pp. 206–215. 20

[25] H. Yu and A. Vahdat, “Combining generality and practicality in a conit-based continuous consistency model for wide-area replication,” in *Distributed Computing Systems, 2001. 21st International Conference on.* IEEE, 2001, pp. 429–438. 20

[26] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A highly available file system for a distributed workstation environment," *Computers, IEEE Transactions on*, vol. 39, no. 4, pp. 447–459, 1990. 22, 32, 33

[27] J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992. 22, 30, 33, 39

[28] D. Ratner, P. Reiher, and G. Popek, "Roam: A scalable replication system for mobile computing," in *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on.* IEEE, 1999, pp. 96–104. 22, 30

[29] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5. ACM, 1995, pp. 172–182. 22, 30, 31, 33, 35

[30] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers, "Flexible update propagation for weakly consistent replication," in *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5. ACM, 1997, pp. 288–301. 22, 31, 35

[31] N. Preguiça, M. Shapiro, J. Legatheaux Martins *et al.*, "Automating semantics-based reconciliation for mobile databases," 2003. 22, 31, 36, 45

[32] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel, "The icecube approach to the reconciliation of divergent replicas," in *Proceedings of the twen-*

tieth annual ACM symposium on Principles of distributed computing. ACM, 2001, pp. 210–218. 22, 30, 31, 36

[33] J. Barreto, "Information sharing in mobile networks: a survey on replication strategies," Citeseer, Tech. Rep., 2003. 22

[34] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002. 23

[35] A. Tanenbaum and M. van Steen, *Distributed systems: principles and paradigms.* Prentice-Hall of India Private Limited, 2003. 25, 27, 29

[36] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009. 28

[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220. 28, 30

[38] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010. 28, 30

[39] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session guarantees for weakly consistent replicated data," in *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on.* IEEE, 1994, pp. 140–149. 29

[40] F. Laux and T. Lessner, “Transaction processing in mobile computing using semantic properties,” in *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on.* IEEE, 2009, pp. 87–94. 31

[41] M. Cart and J. Ferrie, “Asynchronous reconciliation based on operational transformation for p2p collaborative environments,” in *Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on.* IEEE, 2007, pp. 127–138. 31

[42] Z. Abdul-Mehdi, A. Bin Mamat, H. Ibrahim, and M. Deris, “A model for transaction management in mobile databases,” *Potentials, IEEE*, vol. 29, no. 3, pp. 32–39, 2010. 31

[43] S. Phatak and B. Badrinath, “Multiversion reconciliation for mobile databases,” in *Data Engineering, 1999. Proceedings., 15th International Conference on.* IEEE, 1999, pp. 582–589. 31

[44] J. Abawajy and M. Mat Deris, “Supporting disconnected operations in mobile computing,” in *4th ACS/IEEE international conference on computer systems and applications.* IEEE, 2011, pp. 911–918. 31

[45] S. Phatak and B. Nath, “Transaction-centric reconciliation in disconnected client–server databases,” *Mobile Networks and Applications*, vol. 9, no. 5, pp. 459–471, 2004. 31

[46] M. Choi, Y. Kim, and J. Chang, “Transaction-centric split synchronization mechanism for mobile e-business applications,” in *Data Engineering Issues in E-Commerce, 2005. Proceedings. International Workshop on.* IEEE, 2005, pp. 112–118. 31

[47] F. Hupfeld and M. Gordon, “Using distributed consistent branching for efficient reconciliation of mobile workspaces,” in *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on.* IEEE, 2006, pp. 1–9. 31

[48] M. Satyanarayanan, “Scalable, secure, and highly available distributed file access,” *Computer*, vol. 23, no. 5, pp. 9–18, 1990. 33

[49] P. Kumar and M. Satyanarayanan, “Log-based directory resolution in the coda file system,” in *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on.* IEEE, 1993, pp. 202–213. 35

[50] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez, “Reconciliation in the appa p2p system,” in *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, vol. 1. IEEE, 2006, pp. 10–pp. 38

[51] M. Asplund, *Restoring Consistency after Network Partitions.* Linköpings universitet, 2007. 38

[52] G. Kuenning and G. Popek, *Automated hoarding for mobile computers.* ACM, 1997, vol. 31, no. 5. 38

[53] G. Kuenning, “The design of the seer predictive caching system,” in *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on.* IEEE, 1994, pp. 37–43. 38

[54] C. Tait, H. Lei, S. Acharya, and H. Chang, “Intelligent file hoarding for mobile computers,” in *Proceedings of the 1st annual international conference on Mobile computing and networking.* ACM, 1995, pp. 119–125. 39

[55] C. Hu, H. Kim, and J. Hou, "An analysis of the binary exponential backoff algorithm in distributed mac protocols," 2005. 40

[56] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "Macaw: a media access protocol for wireless lan's," in *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 4. ACM, 1994, pp. 212–225. 41

[57] N. Song, B. Kwak, J. Song, and M. Miller, "Enhancement of ieee 802.11 distributed coordination function with exponential increase exponential decrease backoff algorithm," in *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, vol. 4. IEEE, 2003, pp. 2775–2778. 41

[58] J. Deng, P. Varshney, and Z. Haas, "A new backoff algorithm for the ieee 802.11 distributed coordination function," 2004. 42

[59] S. Manaseer and M. Masadeh, "Pessimistic backoff for mobile ad hoc networks," *Al-Zaytoonah University, ICIT*, vol. 9, 2008. 42

[60] W. Scherer III and M. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005, pp. 240–248. 43, 45, 49

[61] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010. 45

[62] W. Scherer III and M. Scott, "Contention management in dynamic software transactional memory," in *PODC Workshop on Concurrency and Synchronization in Java programs*, 2004, pp. 70–79. 49

[63] G. Chockler, D. Malkhi, and M. Reiter, "Backoff protocols for distributed mutual exclusion and ordering," in *Distributed Computing Systems, 2001. 21st International Conference on.* IEEE, 2001, pp. 11–20. 49

[64] J. J. Community, "http://www.jboss.org/hornetq," 2012. 75