View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Newcastle University eTheses

Dynamic Contention Management for Distributed Applications

Matthew Jess Brook

A thesis submitted for the degree of

Doctor of Philosophy

School of Computing Science, Newcastle University

July 2014

Abstract

Distributed applications often make use of replicated state to afford a greater level of availability and throughput. This is achieved by allowing individual processes to progress without requiring prior synchronisation. This approach, termed optimistic replication, results in divergent replicas that must be reconciled to achieve an overall consistent state. Concurrent operations to shared objects in the replicas result in conflicting updates that require reconciliatory action to rectify. This typically takes the form of compensatory execution or simply undoing and rolling back client state.

When considering user interaction with the application, there exists relationships and intent in the ordering and execution of these operations. The enactment of reconciliation that determines one action as conflicted may have far reaching implications with regards to the user's original intent. In such scenarios, the compensatory action applied to a conflict may require previous operations to also be undone or compensated such that the user's intent is maintained. Therefore, an ability to manage the contention to the shared data across the distributed application to pre-emptively lower conflicts resulting from these infringements is desirable. The aim is to not hinder throughput, achieved from the weaker consistency model known as eventual consistency.

In this thesis, a model is presented for a contention management framework that schedules access using the expected execution inherent in the application domain to best inform the contention manager. A backoff scheme is employed to create an access schedule, preserving user intent for applications that require this high level of maintenance for user actions. By using such an approach, this results in a performance improvement seen in the reduction of the overall number of conflicts, while also improving overall system throughput. This thesis describes how the contention management scheme operates and, through experimentation, the performance benefits received.

Acknowledgments

I would first like to thank my PhD supervisor Doctor Graham Morgan for his support and guidance. He has provided encouragement and direction to help me develop the work presented in this thesis.

I would like to thank Derek, Rob and Kamal for their friendship during my study at Newcastle. I am immensely grateful to my Mum and Dad for their support throughout my 8 years of study during undergraduate, masters and PhD at Newcastle University.

I would like to express all my love and appreciation to my wife, Anna. She has provided the support I have needed during this work and has always been there for me.

Contents

1. Introduction **1**

1.1. Distributed Systems . . . 1

1.2. Scalable Data . . . 2

1.2.1. User Intent . . . 4

1.3. Contention Management . . . 4

1.4. Contribution . . . 5

1.5. Publications . . . 8

1.6. Thesis Structure . . . 9

2. Background **10**

2.1. Scalable Data . . . 10

2.1.1. Strong Consistency Models . . . 12

2.1.2. Weak Consistency Models . . . 15

2.1.3. Eventual Consistency . . . 17

2.2. Replication . . . 19

2.2.1. Pessimistic and Optimistic Replication . . . 19

2.3. Replication Techniques . . . 20

2.3.1. The *happens-before* Relationship . . . 21

2.3.2. Logical Clocks . . . 21

2.3.3. Vector Clocks . . . 21

2.3.4. Scheduling . . . 22

2.3.5. Conflict Detection and Resolution . . . 24

2.4. Contention Management . . . 26

2.4.1. Backoff Algorithms . . . 26

2.4.2. Contention Managers . . . 28

2.5. Dynamic Data Access 29
2.6. Related Work 31
2.6.1. Coda 31
2.6.2. Bayou 33
2.6.3. IceCube 34
2.6.4. Cassandra 36
2.6.5. Contention Management Policies 37
2.7. Gap Analysis 40
2.7.1. Application Semantics 40
2.7.2. Reconciliation Scheme 43
2.7.3. Irreconcilable Conflict Management 44
2.7.4. Architecture 44
2.7.5. Consistency 45
2.7.6. Contention Management 45
2.8. Discussion 45

3. Design and Implementation 48

3.1. System Model 48
3.1.1. Client Processes 49
3.1.2. Application Server Processes 49
3.1.3. Database Server Processes 51
3.1.4. Model Discussion 51
3.2. Application Semantics 54
3.3. Implementation 56
3.4. System Model 56
3.4.1. Load Balancer 57
3.4.2. Database 58
3.4.3. Clients 58
3.4.4. Application Server 61
3.5. Semantic Contention Management 64
3.5.1. Contention Manager 64
3.5.2. Dynamic Reconfiguration 66
3.6. Client Injection Rate Modification 67

4. Simulation and Results 69

4.1. Simulation Architecture 69

4.1.1. Data Generation 70

4.1.2. Load Balancer 70

4.1.3. Clients 71

4.1.4. Application Server 71

4.1.5. Database 72

4.2. Results 72

4.2.1. Parameters and Configuration 72

4.2.2. Case Study: E-commerce 74

4.2.3. Case Study: High Frequency Trading 79

4.2.4. Case Study: Network Gaming 85

5. Conclusion 89

5.1. Thesis Summary 89

5.2. Thesis Discussion 92

5.3. Future Work 94

A. Protocol Pseudo Code 102

A.1. Notation 102

A.2. Pseudo-Code 102

A.2.1. Client Protocol 103

A.2.2. Server Protocol 104

A.2.3. Injection Rate Modification 108

List of Figures

2.1. Two execution histories: (a) is sequentially consistent, (b) is not 13
2.2. Two execution histories: (a) is causally consistent, (b) is not 14
2.3. An execution history that adheres to the PRAM consistency model 15
2.4. An execution history satisfying release consistency 16
2.5. An execution history satisfying entry consistency 17
2.6. Identifying the inconsistency window within an eventual consistent replication scheme 18

3.1. Two basic graph configurations 55
3.2. System model 57
3.3. Client Model 60
3.4. Client Workflow 61
3.5. Server Model 63
3.6. Server Workflow 64

4.1. Conflicts: with and without contention management 75
4.2. Conflicts: contention management and reconfiguration 76
4.3. Throughput: with and without contention management 77
4.4. Throughput: contention management and reconfiguration 78
4.5. Conflicts: with and without contention management (reconfiguration), threshold 30 81
4.6. Conflicts: with and without contention management (reconfiguration), threshold 60 82
4.7. Throughput: with and without contention management (reconfiguration), threshold 30 83

4.8. Throughput: with and without contention management (reconfiguration), threshold 60 84
4.9. Conflicts: with and with contention management (high and low latency) . . 86
4.10. Throughput: with and without contention management (high and low latency) 87

List of Tables

2.1. Comparison of background system features 41

4.1. Experiment parameters 74

4.2. Conflicts: Statistics for 50 executions 76

4.3. Conflicts: Statistics for 50 executions 76

4.4. Throughput: Statistics for 50 executions 77

4.5. Throughput: Statistics for 50 executions 78

4.6. Experiment parameters 80

4.7. Conflicts: Statistics for 50 executions, threshold 30 81

4.8. Conflicts: Statistics for 50 executions, threshold 60 82

4.9. Throughput: Statistics for 50 executions, threshold 30 83

4.10. Throughput: Statistics for 50 executions, threshold 60 84

4.11. Experiment parameters 86

4.12. Conflicts: Statistics for 50 executions 87

4.13. Throughput: Statistics for 50 executions 87

A.1. Notations. 102

1. Introduction

1.1. Distributed Systems

The move from centralised computing paradigms to a distributed based system, where processes are separated geographically, is now the basis for a large number of modern applications. Software that once ran in isolation on a single machine is now able to be delivered through a web browser providing a transparent and comparable user experience. A number of fundamental advantages are provided by distributed systems:

- **Scalability**: A distributed system offers the ability to dynamically handle a growing workload. This can be achieved by deploying additional resource to help with processing. The notion of Internet-scale services describes a distributed system that must accommodate millions of users at peak times. Such a system needs to be able to adapt to such growth both in design and performance.
- **Reliability**: Any computing platform wants to provide a reliable service and distributed systems provide means to achieve this. Replication strategies offer the ability for a data set or process to be duplicated such that, in an event of a failure of one service, a second process can transparently step in and maintain performance. This concept of available services allow for guarantees to be provided with regards to the level of performance expected from the system.
- **Performance**: Related to scalability and reliability, distributed applications offer a greater level of performance than their centralised counterparts. A task or number of tasks can be spread across a number of services each performing the same operation. This paralellisation offers significantly increased throughput. A scalable system can deploy a number of a services to cope with an increasing demand, while satisfying the requirements of reliability should one fail.

While distributed systems offer a number of advantages, they are not without their own set of design challenges. Co-ordination in a distributed system is done using message passing between communicating processes. Over any network there exists a delay between the sending of the message and its receipt. Over the Internet this delay, whilst small, is non-trivial and cannot be ignored in the design of a distributed system. In the context of a distributed system with real time requirements, such as virtual environments, this communication latency is a challenge in producing a a realistic simulation.

1.2. Scalable Data

As mentioned in the previous section, replication techniques are a common aspect of distributed systems. Replication sees a collection of data being copied and deployed across a number of processes as part of the distributed system. The reason for using data replication is to provide a greater degree of availability of the data whilst achieving a greater level of performance. As a number of copies of the data exist in the system, should one process with the data fail the whole system does not. Data requests or accesses can simply be redirected to a different replica in the event of failure. These replicas can also be strategically deployed with relation to geographical location to reduce communication latencies.

Early research into replication strategies proposed a pessimistic approach. A primary-copy algorithm sees a replica elected as the primary for all accesses for a given data object. When an object is updated, the primary replica updates this object in all other replicas. Should the primary fail then a new primary is elected from the remaining replicas. Such a technique performs well when communication latencies are low and failures are rare, but, when applied to Internet-scale architectures, provides poor performance due to the synchronization requirements between replicas.

Optimistic replication removes the requirement of synchronisation based on the assumption that conflicts occur rarely. As the system progresses, updates made to shared data objects are propagated in the background and integrated into replicas held by participating sites. Should a conflict be detected then a suitable reconciliation procedure must be executed to resolve it.

The challenge of replication lies in the maintaining consistency among replicas. When a change to a data item occurs it must be sent to all replicas and integrated. In a pessimistic scheme, a high level of consistency is maintained between replicas given the requirement of synchronisation. An optimistic scheme sees a relaxing of this consistency requirement in favour of supporting a more scalable solution. However, when a data object is updated at two or more replicas simultaneously, a conflict occurs. Strategies for resolving conflicts in optimistic systems are typically application dependent, and rely on a level of semantic knowledge of the application to support the reconciliation process.

This thesis focuses on optimistic replication techniques that utilise client-side caching. In such systems, required data objects are replicated and stored locally at the client. Updates are made to the local replica, providing the benefit of offsetting communication latency and allowing for clients to continue processing when disconnected from the server. To maintain consistency, updates must be periodically sent to the server-side, which maintains a master copy of the shared data state.

For a distributed system, the CAP (Consistency, Availability and network Partitioning) theorem [Bre00, GL02] presents an interesting concept. The theorem presents consistency, availability and partition tolerance as properties of a distributed system and, at most, only two of three can be satisfied at one time. As most distributed systems cannot forfeit partitioning, a compromise has to be made in the availability or consistency. Favouring consistency sees a reduction in the availability of services due to state synchronisation, while favouring availability requires a weakening of consistency as state updates are propagated among replicas. Designers often favour availability over consistency in producing scalable systems.

In favouring availability over consistency, the notion of eventual consistency has been introduced as a concept for replicated data. The strong consistency models support a high level of correctness and are applicable for aspects of systems where consistency must be guaranteed such as financial transactions. For data where inconsistency can be tolerated, eventual consistency is suitable. Eventual consistency states that should no new updates be made to a data object, it will eventually return the most up to date value. The concept of an inconsistency window provides a time period in which an incorrect value may be returned for the data object, before the latest update has been applied. Eventually consistency provides a solution for data sets that can tolerate the

inconsistency in favour of supporting scalability and availability.

1.2.1. User Intent

The focus of this thesis is on the intent and action relation requirement that exists between data accesses. In the scenario where a client performs a number of updates to its local replica a degree of intent of the user exists. Consider a simply online purchasing scenario. A user makes a purchase for a bicycle and in addition also buys some accessories to go with the bike; a helmet, set of lights and some mudguards. If each purchase is considered as a separate operation and the order is maintained as per how the user made the purchases, then the purchase for the bicycle is the originating operation of the set and has the greatest level of intent. That is, the subsequent accessory purchases are made on the assumption that the bicycle has been successfully purchased. As such, the intent of the user is preserved in its entirety if all operations are successful. Similarly, if none of the operations are successful then the intent of the user is not maintained. In between these two ends of the intent spectrum there is a another possibility to consider. If the bike purchase fails, then the user is unlikely to want a set of accessories and no bike to use them with. Ensuring that the purchase of the bike is successful has the highest level of importance in this set of operations. A weakening of the intent can be tolerated should the purchase of some or all accessories be unsuccessful. In this scenario, we have a spectrum of intent satisfaction where, depending on the operation type, certain operations must succeed while others can be tolerated to a greater degree should they fail.

1.3. Contention Management

Access to shared data in the context described thus far can be viewed as an issue on contention; a number of processes (i.e. clients) are competing for access to a shared resource where access must be mutually exclusive. A contention manager aims to regulate the access requests of clients in such a way as to provide a level of fairness while managing throughput. A number of contention management policies exist in the literature with each providing a different resolution policy in the presence of a data access conflict. The evaluation of these policies is made by their performance relative to the throughput values achieved.

Contention management policies have seen use in the areas of distributed multiple access and, more recently, transactional memory. Distributed multiple access is deployed in networking protocols such as Ethernet where a number of senders are competing for access to the shared communication medium. If all senders were to send their messages at the same time data collisions would occur and overall throughput would be low. The contention management in this scenario sees senders, in the event of a collision, pause their execution, a process known as backoff. After the backoff period has expired the sender can attempt to access the medium again. This results in more efficient use of shared communication medium as there is a reduction in conflicts and a greater level of throughput achieved. Transactional memory has seen a wide range of contention management policies proposed. Many use the backoff technique seen in distributed multiple access to achieve good performance results.

A number of contention managers have been proposed for use in transactional memory systems, each reflecting the appropriateness of use within specific application types. In a similar manner to contention management in networked hardware, the basic workings are the same: backoff and retry. However, as there is a possibility to succeed (commit) even in the presence of conflict, the contention manager's main role in transactional memory is to determine which transaction to abort and which to allow to continue.

1.4. Contribution

The thesis addresses engineering problems associated with contention management across replicated data in distributed applications. The thesis combines the existing techniques of replication, consistency and contention management to explore their applicability as an approach to contention management that is sufficiently general as to satisfy a wide variety of application types that exhibit a degree of user intent and predictability. The thesis presents work suitable for deployment across standard Internet protocols and may accommodate for a wide variety of application types.

The thesis provides the following contributions:

1. An assessment of contention management in the context of optimistic replication schemes. This is provided in the form of a detailed background chapter, comparative analysis of surveyed systems and gap analysis describing the

motivation and hypothesis behind this work. This background forms the hypothesis that contention management schemes, shown to be successful in areas like transactional memory, can also be applied within weakly consistent systems to achieve similar success. The measure of success for a contention management scheme is a greater degree of access to contentious resources. When considering optimistically replicated distributed systems, contention is seen for concurrent access to replicated data. A contention manager, in this scenario, would be able to mediate concurrent access that can typically lead to conflicts. A reduction in the number of conflicting accesses is desirable as the computation performed to generate the original access is lost. As such, where contention managers have previously been shown to successfully manage access to contentious research, this is taken forward into the domain of distributed systems and optimistic replication.

2. Moving into the application domain presents the opportunity to explore the use of application semantics to inform the contention manager. Existing literature in the optimistic replication domain has already explored the concept of how application semantics can be used to be resolve conflicting operations across a replicated system. This is explored in depth in the background chapter but the essence of techniques is to identify conflicts that can be tolerated given the context of the application or by some form on compensatory action that is close as possible to the original intent of the conflicting operation. Managing this intent is an interesting concept that has not been explored in the optimistic replication domain and, as such, is explored as part of this thesis. By using application semantics that represent the expected execution paths within a system, these are applied to the contention management scheme in an effort to manage contentious resource access, whilst still maintaining the original intent of the operations accessing these resources.

3. These two areas (contention management for distributed applications and the maintenance of intent) are investigated using a client-server test-bed enforcing eventually consistent behaviour for applications that require the retainment of intent at an individual client, but not necessarily across clients. This identifies a class of system common in Internet applications where clients operate autonomously (not collaboratively) in their dealings with a server. The architecture of the system mimics that described in the literature for existing

optimistically replicated and distributed systems. Taking this forward a contention management scheme is introduced that makes use of the semantic application knowledge to best manage the contention in the system. The contention management scheme is demonstrated to afford the ability to autonomously alter its behaviour over time to best reflect changing interaction patterns of clients and servers. This allows the generality of the solution to be suitable for a wider variety of application types, and to prevent changes in communication patterns from having a lasting adverse impact on overall performance. This is an important aspect, as this work shows that contention management itself does not have to be as bespoke a solution as described in transactional memory literature.

4. Evaluation of the system is made through simulation and describes how the use of both contention management in the context of distributed applications and application semantics can be used to better manage access to shared data in an optimistic replication scheme. As mentioned, the success of a contention manager is measured as providing a greater degree of access to contentious resources. This is no different when applied to the context of optimistic replication and as such an evaluation of the proposed system investigates whether this is achieved. By comparing the number of conflicted operations (those that could not be maintained by the system due to concurrent access with other updating clients) when contention management is and is not used provides a metric with which to determine the effectiveness of the proposed solution in this thesis. A greater degree of throughput can also be achieved by contention managers and is another metric that is investigated. Finally, through the use of application semantics, the contention management scheme becomes self regulating ensuring that changes in the application semantics are handled dynamically. This additional extension to the protocol is investigated in the results to determine the effectiveness of this as an approach.

5. To validate the generalisation of this approach experiments were conducted using three case studies that reflect popular application types: (a) e-commerce - a typical client-server relation reflecting shopping cart like properties; (b) financial trading - a high throughput environment (high frequency trading - HFT) with a requirement on state access timeliness and high overall throughput; (c) massively multiplayer online gaming (MMOG) - an environment where the approach is adapted to allow

integration into protocols governing inter-player communication (i.e., accesses spans client requests).

Although not the main focus of this thesis, it should be pointed out that the overall solutions derived for the case studies (e-commerce, HFT and MMOG) are research advances in their own individual fields of study, and have been published as such.

1.5. Publications

The following publications contain work relating to the contribution of this thesis, along with other projects the author has worked on:

- Yousef Abushnagh, Matthew Brook, Craig Sharp, Gary Ushaw, Graham Morgan. Liana: A Framework that Utilizes Causality to Schedule Contention Management across Networked Systems. *On the Move to Meaningful Internet Systems: OTM 2012,* volume 7566 of *Lecture Notes in Computer Science,* pages 871-878. Springer Berlin Heidelberg, 2012.

- Kamal Solaiman, Matthew Brook, Gary Ushaw, Graham Morgan. A Read-Write-Validate Approach to Optimistic Concurrency Control for Energy Efficiency of Resource-Constrained Systems. In *Proceedings of the 9th IEEE International Wireless Communications and Mobile Computing Conference*, IWCMC '13, pages 1424-1429, 2013.

- Matthew Brook, Craig Sharp, Graham Morgan. Semantically Aware Contention Management for Distributed Applications. *Distributed Applications and Interoperable Systems,* volume 7891 of *Lecture Notes in Computer Science,* pages 1-14. Springer Berlin Heidelberg, 2013.

- Matthew Brook, Craig Sharp, Gary Ushaw, William Blewitt, Graham Morgan. Volatility Management of High Frequency Trading Environments. In *Proceedings of the 15th IEEE Conference on Business Informatics,* CBI '13, pages 101-108, 2013.

- William Blewitt, Matthew Brook, Graham Morgan. Towards Consistency of State in Massively Multiplayer Environments through Semantically Aware Contention Management. In *IEEE Transactions on Emerging Topics in Computing,* 2014.

1.6. Thesis Structure

The thesis is structured as follows:

- Chapter 2 covers the background and related work. This includes a discussion of scalable data with a focus on strong and weak consistency models including eventual consistency. The topic of replication is covered in the context of optimistic replication techniques such as conflict detection and resolution. The background concludes with a discussion of existing contention management techniques seen in networking protocols and transactional memory. The case studies that this work focuses on are introduced and discussed in the context of shared data access and contention they typically see and how the issues of predictability and semantic knowledge are important considerations in these environments. The related work discusses some existing replication systems and the techniques they use to manage shared data access.
- Chapter 3 describes the design of the contention management framework in the form of a system model and protocol design. This includes a number of pseudo-code implementations describing the operation of the different processes involved.
- Chapter 4 provides a description of the implemented simulation used to test the performance of the proposed approach. This section also presents performance results gathered from the simulation. This covers results gathered relating to the case studies introduced in chapter 2.
- Chapter 5 gives the conclusion to this thesis and provides possible future work.

2. Background

This chapter contains background information relating to scalable data, dynamic data access and contention management. These three areas provide the foundation for the contribution of this thesis. Providing scalable data solutions addresses topics of consistency and replication. Managing consistency in a distributed system is first explored ranging from strict to weak consistency models. This leads on to eventually consistent systems, where the causality requirements are relaxed to afford a greater degree of availability while supporting networking partitioning. The issue of consistency is an important consideration when looking at the topic of replication, where multiples copies of the same data exist to improve availability. Pessimistic replication is first briefly discussed, to provide the counterpoint to the more universally accepted paradigm of optimistic replication. Techniques required to manage the issues of scheduling and conflict detection are discussed with regards to providing an optimistically replicated system. Dynamic data access discusses the topics of accesses and predictability. The access requirement is a specific focus of this thesis; the execution of an operation has a purpose and intent attached to it by the user. Ensuring that this is maintained, by utilising a user's predictable behaviour, in the context of a large scale distributed system, is addressed by this work. Finally, the area of contention management is considered, with examples seen in networking and more recently within the domain of transaction memory, where high volumes of access requests have to be scheduled given a limited set of resources.

2.1. Scalable Data

Scaling a distributed system requires service provisioning to not only handle an increasing volume of requests, but to also ensure that the data associated with those requests is available when considering network partitions or process failure. A typical

approach to this problem is to make copies of data, a technique called replication. These copies form a number of replicas that can be strategically placed to increase the availability of data or to offset network latency with regards to geographical placement or available bandwidth and connectivity. The cost of replication is ensuring that these replicas maintain consistency when changes are made at one.

As presented in Brewer's CAP theorem [Bre00, GL02], there is a trade off in managing the consistency (C) and availability (A) of a distributed system in the presence of network partitions (P). The trade off comes from being able to satisfy at most two of the requirements at the expense of the third. As an example, consider two nodes separated by a network partition. If one of these nodes updates their local replica, this will cause the other node to become inconsistent, sacrificing consistency. If we wish to maintain the overall consistency then we must make the inconsistent node unavailable, sacrificing availability. To preserve both consistency and availability, the nodes must be able to communicate, at which point network partitioning is sacrificed. In the majority of distributed systems, is is not possible to forfeit partitioning and so compromises must be made in how availability and consistency are managed.

The ACID mnemonic [Gra81, HR83] describes four properties primarily relating to the processing of database transactions. These are:

- **Atomicity**: The "all or nothing" property; if any part of the transaction fails, the entire transaction fails. If the transaction fails, the state of the database remains unchanged, as if the transaction never executed.
- **Consistency**: A successful transaction moves the database from one consistent state to a new and valid consistent state.
- **Isolation**: The execution of transactions occurs concurrently and the isolation property requires that the state achieved is the same as if having executed each transaction in sequence.
- **Durability**: Upon completion, the changes made by a transaction are committed to the database and will remain even in the event of a crash or power loss.

These ACID properties provide a strong consistency guarantee in the presence of no network partitions. Network partitioning can be addressed through the use of a

commitment protocol such as two-phase commit [BHG87]. The two-phase commit protocols requires that all databases involved are available for the duration of both phases to commit changes. In providing the C and P of CAP, systems that adhere to the ACID properties provide strong consistency guarantees at the expense of availability.

An alternative design paradigm to ACID, aimed at relaxing consistency requirements to afford greater availability, is the BASE mnemonic: Basically Available, Soft state, Eventually consistent [FGC+97, Pri08]. A system exhibiting BASE properties can tolerate partial failure and inconsistency in data on the assumption that replicated data will become eventually consistent. As such, the consistency of the system can be seen as in a constant state of flux, as replicas converge and diverge towards and from a globally consistent state.

These two designs, ACID and BASE, are not mutually exclusive; aspects of both methodologies can be present in a single system. Typically, an ACID approach using transactions would be deployed in parts of the system that require the greatest level of consistency, while a BASE approach would be better suited to aspects of the application that can tolerate inconsistency for the increased availability. This choice between availability or consistency may occur many times in the same system, and vary depending on the operation type or user requirements. Systems like Cassandra, discussed in section 2.6.4, address this issue by providing tunable consistency where the consistency of each operation can be specified depending on specific client requirements.

For shared data access and replication, consistency among replicas must be managed to ensure overall system correctness. Updates made at one replica must be disseminated and integrated at all other replicas to maintain an overall consistent system state. There exist a number of consistency models that provide differing levels of consistency guarantees. These models are typically applicable with regards to the required correctness criteria of the implementing application. Consistency models broadly fall into one of two categories: strong or weak. Within each category there exist a range of models that present varying levels of consistency given specific operational semantics.

2.1.1. Strong Consistency Models

Strict consistency is the most stringent consistency model, requiring that a read to a shared data item x always returns the result of the most recent write to x. The

implementation of a strict consistency models requires absolute global time that, while is achievable in uniprocessor systems, cannot be achieved in a distributed system as multiple operations (read or write) can be performed simultaneously.

Sequential consistency is a weaker model than strict consistency, defined as follows:

"… *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*" [Lam79]

Figure 2.1 illustrates two execution histories: (a) is sequentially consistent, while (b) is not. In part (a), process 1 writes to the shared object x the value 1 followed by process 2 writing to x the value 2. Processes 3 and 4 read the values 2 and 1 consistently with respect to each other. In part (b) the ordering is not sequentially consistent. Processes 1 and 2 perform the same write operations but the processes 3 and 4 read inconsistent values for x, an ordering that is not sequential.

(a)

P_1:	W(x) 1				
P_2:		W(x) 2			
P_3:			R(x) 2		R(x) 1
P_4:				R(x) 2	R(x) 1

(b)

P_1:	W(x) 1				
P_2:		W(x) 2			
P_3:			R(x) 2		R(x) 1
P_4:				R(x) 1	R(x) 2

Figure 2.1.: Two execution histories: (a) is sequentially consistent, (b) is not

A third consistency model, linearizability [HW90], is weaker than strict consistency but stronger than sequential consistency. Linearizability extends the definition of sequential consistency by time stamping each operation. For two operations, a and b, if the timestamp of a is less than the timestamp of b then a must occur before b in the ordering. Part (a) of figure 2.1, while sequentially consistent, could not be considered linearizable. The final two reads of processes 3 and 4 would have to return the value 2 as the write performed by process 2 is ordered before these reads. Linearizability is weaker than strict consistency, as it assumes the use of loosely synchronised clocks but cannot provide the same requirements of an absolute global time.

Causal consistency [ANB$^+$95] (based on Lamport's concept of potential causality [Lam78]) on is a weakening of sequential consistency that identifies operations that may

be causally related from those that are not. For example, a read of a an object x followed by a write to an object y can be considered causally related because the value read of x may have been required to compute the value written to y. Operations that are not causally related are considered concurrent. As such, writes that are considered causally related must be seen by all processes in the same order while concurrent writes can be seen in different orders among processes.

P_1: W(x) 1			W(x) 3	
P_2:	R(x) 1	W(x) 2		
P_3:	R(x) 1			R(x) 3 R(x) 2
P_4:	R(x) 1			R(x) 2 R(x) 3

(a)

P_1: W(x) 1			
P_2:	R(x) 1	W(x) 2	
P_3:			R(x) 2 R(x) 1
P_4:			R(x) 1 R(x) 2

(b)

Figure 2.2.: Two execution histories: (a) is causally consistent, (b) is not

Figure 2.2 illustrates two execution histories: (a) is causally consistent, while (b) is not. In part (a), the first write of process 1 (W(x) 1) is causally related to the write made by process 2 (W(x) 2). This is because process 2 first observes the write of process 1 when reading x. Processes 3 and 4 read the value x in different orders which would violate sequential consistency but is fine under causal consistency. Part (b) cannot be considered causally consistent as processes 3 and 4 observe the value of x as 1 although the writes of processes 1 and 2 (W(x) 1 and W(x) 2 respectively) could be considered causally related.

A further weakening of consistency leads to the Pipelined Random Access Memory (PRAM) consistency model [LS88] (also known as the First In First Out consistency model). In this model, the causality requirement is removed to facilitate an easier implementation when compared with the overhead of tracking causally related writes in causal consistency. PRAM consistency requires that writes performed by a single process must be seen by all other processes in the same order as which they were issued while writes from different processes may be seen in different orders by different processes. In part (b) of figure 2.2, what was previously considered a violation of causal consistency would be acceptable under PRAM consistency. Figure 2.3 additionally illustrates an execution history that adheres to PRAM consistency. Both processes 3 and 4 see the writes of process 2 in the order they were issued but view the write of process 1 in a different order. To be considered causally consistent, processes 3 and 4 should not be able to observe the write of process 1.

P_1:	W(x) 1							
P_2:		R(x) 1	W(x) 2	W(x) 3				
P_3:					R(x) 2	R(x) 1	R(x) 3	
P_4:					R(x) 1	R(x) 2	R(x) 3	

Figure 2.3.: An execution history that adheres to the PRAM consistency model

2.1.2. Weak Consistency Models

The strong consistency models presented in the previous section all require, in some form, the other processes to be made aware of updates made by other processes. It may not be required to have all processes receive every update, rather that the most recent update for a shared object is relevant. Through the use of a synchronisation operation each process can be guaranteed to be consistent with respect to the most recent synchronisation operation. The weak consistency model [DSB86] provides the following three properties to describe a weakly consistent data store:

1. Accesses to synchronisation variables sequentially consistent among processes.
2. No accesses to a synchronisation variable can be allowed before all previous data accesses have been performed.
3. No data accesses are allowed until all previous accesses to synchronisation variables have been performed.

For example, if we consider a critical section in which a process updates a data structure, a synchronisation primitive can be used before entry. This initial synchronisation ensures that the process has a consistent view of the data, and similarly, when other processes come to enter that same critical section. The burden of managing consistent access now lies with the programmer identifying synchronised accesses.

In the weak consistency model, a synchronisation variable does not distinguish between a process either starting to read a shared data item or finishing writing to the shared data item. The release consistency model [GLL+90] extends the weak consistency model allowing for more efficient implementations by exploiting additional information about shared accesses. To this effect, two new primitives are introduced:

- acquire - informs the data store that a process wishes to enter the critical section.

- release - informs the data store that a process has left the critical section.

Figure 2.4 presents an example execution history using the above primitives. The Acq(L) and Rel(L) labels signify the acquire and release primitives to a shared lock L respectively. Process 1 acquires the lock, writes to x and releases the lock. Process 2 acquires the lock and reads the value of x as 2, the latest write of process 1, and releases the lock. Finally, process 3 reads the value of x as 1 without acquiring the lock first. The execution of the acquire operation guarantees that the local version of the data in a process is made consistent with the remote copy. When the release operation is called at the end of the section of execution, the changes made in the critical section will be propagated to other processes. Process 3 reads an inconsistent value of x because it has not acquired the lock which would have ensured x was consistent. The release operation on the lock would initiate the propagation of the updated state of x to process 3 but given the delay in communication, without using the acquire operation, we cannot guarantee consistency in of x in process 3.

P_1: Acq(L) W(x) 1 W(x) 2 Rel(L)
P_2: Acq(L) R(x) 2 Rel(L)
P_3: R(x) 1

Figure 2.4.: An execution history satisfying release consistency

Through the correct use of the acquire and release operations, execution results will be the same as a sequentially consistent shared data store. As such, the burden lies with the programmer in ensuring these primitives are correctly used to maintain consistency. The above description focuses on a specific form of release consistency called eager release consistency, where updates are propagated when the release operation is executed. The alternative, lazy release consistency, defers the update until a subsequent acquire is performed by a process. In certain circumstances it has been shown that lazy release consistency performs better than eager release consistency [KCZ92].

The final weak consistency model of this section is the entry consistency model [BZ91]. In this model, the acquire and release operations explicitly state which shared data items are being locked. Each shared data item is associated with a lock variable unique to that item. The tradeoff for the increased overhead of additional synchronisation variables,

and programming effort to manage these, is the increased parallelism. For example, two processes could access an array of data and synchronise using locks belonging to the elements of the array rather than a single lock belonging to the whole array. In release consistency, acquiring the lock requires that the system determines the required variables for use in the critical section that need to be updated. Entry consistency removes this requirement, as the synchronisation is made explicit given the lock per shared variable. Entry consistent also distinguishes between exclusive and non-exclusive accesses such that non-exclusive accesses can be granted to a shared variable concurrently.

P_1:	Acq(Lx)	W(x) 1	Acq(Ly)	W(y) 2	Rel(Lx)	Rel(Ly)				
P_2:					Acq(Lx)		R(x) 1	R(y) 1		
P_3:									Acq(Ly)	R(y) 2

Figure 2.5.: An execution history satisfying entry consistency

Figure 2.5 shows an execution history demonstrating the shared object based locking of entry consistency (this execution assumes competing accesses). The acquire operation now distinguishes which shared object is being accessed; Acq(Lx) signifies an acquire to the synchronisation variable Lx relating to the shared object x. Similarly, the release operation is related to the previous synchronisation variable that was acquired. Process 1 acquires the lock for shared object x, writes to it, acquires the lock for shared object y, writes to it and releases both locks. Process 2 is able to acquire the lock for x as it has been released by process 1. This results in the local version of x for process 2 is updated to the latest value (when process 1 wrote to x). The read by process 2 to variable y results in an inconsistent value. This is because process 2 has not acquired the lock for y and so there is a guarantee about the consistency. Process 3 is able to acquire the lock for y and read the consistent value.

2.1.3. Eventual Consistency

Eventual consistency is a specific form of weak consistency, predominantly explored within managing consistency for large scale replicated systems, that is defined as:

"... if no new updates are made to [an] object, eventually all accesses will return the last updated value." [Vog09]

With strict consistency subsequent accesses to that object (i.e. shared variable) must return the latest value in all processes. Weak consistency provides an inconsistency window: the period between the update and the time when it is guaranteed that a process will see the updated value. Not considering failure, this inconsistency window can be reasoned about given a number of contributing factors such as communication delay, replica load and the number of replicas in the replication scheme. Figure 2.6 shows an execution history involving three replicas that illustrates eventual consistency. At Replica 1 a client writes to data item x the value 1. In Replicas 2 and 3 two clients read the value of x as 0. Outside of the inconsistency window, all replicas return the most up to date value of x.

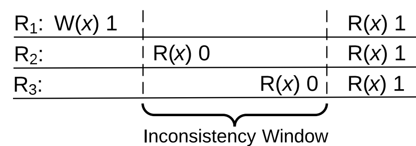

Figure 2.6.: Identifying the inconsistency window within an eventual consistent replication scheme

The above example works when clients are interacting with the same replica for all operations. Requiring a client to interact with the same replica for all operations may not always be possible and negates some of the benefits that replication provides. Managing the consistency of a per-client basis such that a user perceives their own updates when moving between replicas is managed using the notion of a session guarantee. As described in [TDP+94] four session guarantees exist:

- **Read Your Writes**: The effect of a write to a data item will always be seen by any subsequent read to the same data item for a given client. This avoids the issue of confusion when users and applications make an update to a database only to find the change missing when immediately reading.
- **Monotonic Reads**: Having observed a value for a shared data, this guarantee ensures that the user will only ever observe the same value or a more up to date version for the data item.
- **Writes Follow Reads**: Any writes are ordered after writes whose effects were seen by previous read operations. This guarantee differs to the previous two as it affects users outside of the session ensuring that they also see the same ordering of writes.

- **Monotonic Writes**: Writes must follow previous writes within the session. A write is only made to a database if all previous writes in the session have also been applied to the database.

The purpose of these session guarantees is to present applications with a view of the database that is consistent, even in the event of reading from a number of potentially inconsistent sources (i.e. replicas).

2.2. Replication

Data replication sees a number of copies of the same data being made available at any one time. The aim of data replication is to enable higher availability and performance. Availability is improved, as access to data can still be made in the event that some of the replicas are unavailable. Increased performance comes in the form of a reduction in latency and an increase in throughput. Users can access nearby replicas and through the use of a number of replicas, data can be provided simultaneously. The cost of replication comes from managing the consistency between a number of replicas such that an overall consistent view of the data is presented.

2.2.1. Pessimistic and Optimistic Replication

Traditional approaches to replication are termed pessimistic and aim to provide a single-copy consistency model [HW90] such that there is a perception of a single, highly available version of the data. Under a pessimistic replication scheme, access to a replica is blocked until it can be proved to be consistent with respect to all other replicas. An example of a pessimistic scheme is primary-copy algorithms [BHG87] whereby all access to a particular object are managed by a designated replica. Changes made to the objects at a primary replica are then propagated and applied at other replicas. Should the primary replica fail then a new primary will be elected. Such schemes perform well in local-area networks where network latencies are low and failures are uncommon.

Saito and Shapiro [SS05a] present three key reasons why pessimistic algorithms cannot be applied to Internet-scale environments. These are:

1. **Connectivity and latency**: Latency is critical when considering the communication overhead required for the replicas to receive state updates from

each other. Similarly, connectivity can often be intermittent, especially in the context of mobile computing, and as such a a pessimistic algorithm would block when trying to contact an unavailable site.

2. **Scalability**: In the presence of frequent updates to replicas, it becomes difficult to manage a large scale pessimistic replication scale as availability and throughput suffer as the number of replicas increases.

3. **Collaboration**: A pessimistic approach to the domain of collaborative working such as document editing or software engineering would not be feasible. Individuals often work in isolation and it is better to manage the occasional conflicts that occur rather than deny access when one person is editing the document.

Optimistic replication schemes have proved to be popular in the context of wide-scale and mobile environments. Whereas in pessimistic approaches where synchronisation is required prior to data access to guarantee consistency, an optimistic approach does not require this prior synchronisation. This is based on the assumption that conflicts occur rarely (similar to the principles behind optimistic concurrency control). To maintain consistency among the replicas, updates are propagated in the background, allowing replicas to converge back towards a globally consistent state. The benefits lie within the increased ability for scalability and availability. As there is no longer need for prior synchronisation between replicas, a larger number of replicas can be deployed and access to these is flexible with regards to the connectivity requirement typically imposed when using a pessimistic scheme. However, the drawbacks of optimistic approaches lie within the challenges of managing diverging replicas and the conflicts that occur from concurrent operations. As such, optimistic schemes are typically only considered for applications where conflicts are both rare and, in the event of conflicts, can be tolerated or reconciled easily.

2.3. Replication Techniques

This section will discuss important techniques used as part of optimistic replication systems. Operations are submitted independently at different replicas and require scheduling and conflict detection. The ordering between these operations is often used to aid scheduling such that the intent and expectation of the users are maintained.

2.3.1. The *happens-before* Relationship

Given unpredictable delays in communication, guaranteeing a total ordering between events is not possible. The notion of *happens-before* (denoted by "→") introduced by Lamport [Lam78] provides a partial ordering such that, given two events α and β, α *happens-before* β if:

1. Both α and β are events in the same process and α occurs before β.
2. The event α signifies the sending of a message by one process and β signifies the receipt of that message in another process.
3. Given $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$.

If the *happens-before* relationship cannot be applied to two events then these events are said to be concurrent. This technique is used in optimistic replication for scheduling (determining operation orderings when producing an update schedule) and conflict detection (whether two events are concurrent and conflict).

2.3.2. Logical Clocks

Also known as a Lamport clock, a logical clock is a monotonically increasing counter. This timestamp mechanism can be used to capture the *happens-before* relationship. The correctness criteria for a system of logical clocks is the *Clock Condition*: for any events α, β: if $\alpha \rightarrow \beta$ then $C(\alpha) < (\beta)$. Informally, if event α occurs before β then α should happen at an earlier time than β. The converse of this cannot hold as it would imply that any two concurrent events occur at the same time. To implement the Clock Condition, a system must follow these set of rules:

- A process P_{i} increments its logical clock C_{i} when an event (sending / receiving a message or an internal event) occurs.
- When a process sends a message, it attaches the logical clock value to the message.
- On receiving a message, the receiving process sets its own logical clock to be greater than the maximum of both the received clock and its own logical clock.

2.3.3. Vector Clocks

A vector clock (VC), also known as a vector clock [PPR+83, Fid88, Mat89], is a structure that can be used to both determine the *happens-before* relationship and detect

causality violations. The vector clock is an array of n logical clocks, where n is the number of replicas in the system. Each element of the array is a logical clock representing the state of the replicas in the system. Each replica r manages a vector clock VC_r. The vector clock of a replica is updated as follow:

- A replica updates its own vector clock entry whenever it experiences an event (send, receive or internal). For example, if an operation was submitted at replica r then $VC_r[r] = VC_r[r] + 1$.
- When a replica sends a message, it attaches the vector clock to the message. For a message m, $m.VC$ represents that vector clock.
- When a replica receives a vector clock from another replica, its own vector clock is updated as such: $VC_r = max(VC_r, m.VC)$. That is, for every element of the replica's vector clock, the maximum value is taken from the received vector clock (in the message) and its own vector clock.

2.3.4. Scheduling

Updates applied at one replica are propagated to other replicas in the background. The result is that operations are not always received in the same order at all replicas. The replicas need to agree on an ordering of operations that produces an equivalent end state at all replicas while also maintaining the requirements of the users with regards to expectation and intent. The mechanisms used to manage these orderings are termed the scheduling policies. Approaches to scheduling can broadly be categorised as either syntactic or semantic:

Syntactic scheduling

Syntactic scheduling aims to maintain the *happens-before* relationship so as to not have an objects state appear to revert to historical versions (roll back). A typical syntactic scheduler will use a timestamp approach such as logical clocks to order operations as described in section 2.3.2.

Semantic scheduling

Semantic scheduling aims to exploit application-specific knowledge to aid the scheduler in determining a more optimal schedule than that of a syntactic approach. Semantic

techniques typically consider the semantic relationship between a pair of operations, while either maintaining and extending the *happens-before* relationship or replacing it. Semantic scheduling approaches are outlined below:

- **Commutativity**: If two operations commute (the result of the operations is identical when ordered either way round) then this can be taken into advantage by a scheduler when attempting to produce an optimal schedule. A basic example would be two read operations to a shared variable.

- **Operational Transformation**: This technique was developed for collaborative editors as described in [EG89, SE98]. In this technique an operation is applied locally at the replica and propagated to all other replicas. All other replicas apply the updates as received and do not reorder updates. The result is that two replicas may have the same set of operations but applied in different orders (having received them in different orders, not through explicit reordering). Operational transformation defines rules on how to rewrite editing operations for concurrent operations such that the intent of operation (the required changes to the text) are identical regardless of the order that they are applied at a replica. To illustrate the technique, consider a text editing scenario with two users editing a string in the document "replica". User 1 inserts a character "h" at position 1 while user 2 deletes the character "c" at position 5. Executing the insertion at one replica will produce "replica" but the intent of the delete operation made by user 2 is no longer maintained as the character as position 5 is now "i". Using operational transformation would see the position of the delete statement incremented to correctly remove the character "c". An in-depth review of operational transformation techniques is beyond the scope of this work.

- **Optimisation**: An approach introduced in the IceCube system, that is further discussed in section 2.6.3, which aims to define a set of constraints that exist between operations. With these constraints, IceCube aims to produce an optimal schedule containing an ordering of operations that satisfies the greatest number of compatible operations.

The main comparison between syntactic and semantic scheduling approaches is the implementation complexity. A syntactic scheduler, at its most basic, only requires a well-understood implementation of the *happens-before* relationship. Such

implementations can be made generic and are applicable across a wide number of systems. The downside is the potential for unnecessary conflicts that, with a level of semantic and application-knowledge, could otherwise be reordered and avoided. An example of an unnecessary conflict can be seen in the following scenario: an online shopping site has stock remaining for one purchase. Three requests are submitted: (1) item purchase, (2) item purchase, (3) item return. If these requests are processed in the order 1,2,3 then request 2 will fail. Semantic techniques can avoid this issue by reordering the requests 3,1,2 satisfying all users. The overhead lies in the initial implementation requirements to produce application-specific mechanisms to semantic scheduling, as these techniques are not generic. Given the example, the system would need to be able to determine that request 3 is a return and that it should be ordered before any purchases for the same stock item.

2.3.5. Conflict Detection and Resolution

A conflict occurs when an operation fails to satisfy its precondition given the current state of the replica. An operation that has tentatively been applied at a local replica may become conflicted due to scheduling algorithms determining a different ordering of operations. The aim of a scheduling algorithm is to reduce the number of conflicted operations in the resulting schedule so as to satisfy and maintain the intent of the original users. Inevitably, there will be a number of operations that cannot be applied, given the current schedule, and so mechanisms are needed to both detect when a conflict has occurred and decide how to resolve the conflict. Similar to scheduling, conflict detection can be broadly classified into syntactic and semantic techniques.

The simplest solution to conflict management is to simply ignore conflicted operations. A system implementing the Thomas's Write Rule Tho79 is considered eventually consistent, but does not provide any mechanisms for detecting or handling conflicts; they are simply discarded. By using a timestamp-based approach, when writing to an object an update whose timestamp is lower than another update does not need to be performed. The reasoning for this is that, should the update with the lower timestamp be performed, it will eventually be overwritten by the update with the greater timestamp anyway. The impact of such a scheme is the notion of lost updates, the intent behind the discarded update is no longer represented in the system.

An extension to the Thomas's Write Rule is the Two-timestamp algorithm GHOS96 that

enables conflicts to be detected. For every object, two timestamps are maintained; one timestamp shows the newness of the object and a second timestamp indicates when the object was last updated. When two replicas synchronise they exchange timestamps. A conflict is detected if the second timestamps (when the object was last updated) are not equal between replicas. If the newness timestamp of one replica is less than the others, then the objects only needs to be updated at the original replica. The drawback of this approach is the potential for false conflicts when operating with more than two replicas.

By leveraging semantic knowledge of the system and updates, there may be a potential solution to the conflict that maintains both updates or preserves the intent in some form, should one update not be successful. For example, a version control system uses semantics to determine a true conflict. Multiple users can create different files concurrently inside a directory, but updates to the same file will cause a conflict. IceCube and Bayou, discussed in section 2.6, allow the application developer to program explicit preconditions to determine whether an operation is in conflict. The drawback of semantic approaches is the requirement to program preconditions for all operation types in the system. Similar to semantic scheduling, these will be typically application-specific and cannot be created once and used across a range of systems, in contrast to syntactic approaches.

Once a conflict has been detected it must be handled in an appropriate manner. The Thomas's Write Rule will simply ignore the conflict, but other approaches will either attempt to rewrite the update so that is it compatible with the schedule or abort the update and notify the user. A manual approach to conflict resolution will require the user to make the decision on how to handle the conflict. Going back to our version control example, if two users have updated the same file locally, when attempting to update the file, a conflict will be detected and the user will be presented with the the option to merge the files or to handle the conflict otherwise. Automatic conflict resolution mechanisms require no user intervention to handle conflicts but rather run existing procedures to combine the conflict objects creating a new object. A third approach requires users to provide alternatives to the original update that will still satisfy their intent. In this scenario, if the original update is found to be conflict then a number of other options can executed in the attempt to find a satisfactory result that does not conflict. An example of this is a room booking application where the room can only be booked by one person in a given time slot. A conflict occurs if two users attempt

to book the room concurrently. In this scenario the users would provide a second choice of room that would still satisfy their requirements and will avoid the original conflict. The drawback here is that there is no guarantee the alternative room will also be available. Multiple alternative rooms could be a potential solution but this puts the burden of the implementation with the user, which would typically be avoided.

2.4. Contention Management

Conflicts over shared-resources result in a level of contention that requires mechanisms to best manage and resolve access so that the properties of fairness, timeliness and correctness are satisfied for all competing processes. Techniques to manage contention are often seen in the context of concurrency control solutions with solutions such as locks, semaphores, transactions or scheduling algorithms. The family of techniques known as backoff algorithms are of particular relevance to this work, as they form the basis for the type of contention management used in the protocol. This, coupled with existing contention manager architectures, form the discussion for this section.

2.4.1. Backoff Algorithms

In the context of computer networking, contention arises when processes compete for access to the communication medium. The result of two or more processes attempting to transmit data of the medium is a collision where all transmission is unsuccessful. A process will attempt to retransmit the packet, and without a mechanism in place to manage access, will subsequently result in collision and ultimately deadlock.

A technique known as backoff can be used to reduce the number of collisions and address the issue of contention for the medium. The aim of the algorithm is to throttle the access rate of processes to the medium while attempting to arrive at an acceptable balance between all competing processes. An example of an early backoff algorithm is the Binary Exponential Backoff (BEB) algorithm. This is an exponential backoff algorithm that is used in network communications such as Ethernet, to schedule packet retransmission after a collision has occurred. After a collision, the senders will wait a random number of time slots, listen to the channel and, if idle, retransmit. The number of time slots to wait is chosen an a number between 0 and 2^n- 1, where n is the number of collisions so far for this packet. After the first collision, the sender will choose 0 or 1 time slots to wait. If

there is a second collision, a number of time slots between 0 - 3 is chosen to wait. This range, called the contention window, will continue to increase until either the packet is successful transmitted or if the number of collisions reaches ten. Upon successfully transmitting after a collision, the sender will reduce their contention window size to the minimum size. In the context of Ethernet, after sixteen collisions the sending process aborts. As such, access to the medium is controlled by making processes wait before they can attempt to send essentially creating a schedule of access. While there is no guarantee senders will pick different contention window values, the probability they will collide again reduces after successive collisions. The performance of this algorithm lies in finding a balance in managing the size of the contention window. With a large number of senders and a small contention window size there will be a greater risk of collisions. However, a large window when there are fewer senders will lead to unnecessary delay and under-utilisation of the medium. Issues of fairness have been raised with the BEB algorithm in that a sender who is "more" backed off (has a larger contention window) is a lot less likely to be able to send compared to another sender with a much smaller contention window. Without a maximum contention window size, there is the potential for a single sender to capture control of the channel as their window size will always remain to a minimum while other senders will continue to grow ever larger [BDSZ94].

An alternative to the BEB algorithm, called the Multiplicative Increase, Linear Decrease (MILD) algorithm, addresses the issues of unfairness for senders with large contention windows [BDSZ94]. In this scheme, rather than decrease the contention window size to a minimum after a successfully retransmission, the window size is decreased by one (linear). When the sender's transmission collides the window size is increased by a factor of 1.5 (multiplicative). The resulting changes still favour reasonable contention window growth when contention is steadily high and also avoiding the need to repeat the window size escalation after every successful transmission when compared to BEB. The MILD algorithm does not provide any significant improvement to throughput compared to BEB in environments where contention is low, as resetting the contention window size to a minimum does not impact on performance. In scenarios where the number of senders changes from high to low, MILD cannot adjust the contention window size quickly enough to the linear decrease mechanism.

To address issues with MILD, the Exponential Increase, Exponential Decrease (EIED) algorithm both increases and decreases the contention window size exponentially for

collisions and successful retransmission, respectively [SKSM03]. After a sender is involved in a collision, the contention window size is increased by a factor r_{r}. After successfully sending, the contention window is decreased by a factor of r_{D}. The performance of EIED is affected by the choice of values for r_{r} and r_{D}. Performance of EIED is seen to provide a greater throughput to both BEB and MILD given a number of different simulation scenarios (large number of senders, varying arrival rate) [WCP+02, SKSM03].

In summary, these algorithms, also termed a Distributed Coordination Function (DCF) in the domain of wireless routing, only differ in the way they adjust the contention window size after collisions or successful transmission. The result is a schedule that provides coordination between a number of nodes allowing access to a shared resource. This coordination allows for a greater level of throughput and utilisation of the medium; without it senders would not be able to effectively communicate due to constant transmission collisions.

2.4.2. Contention Managers

More recently the issue of contention has been addressed in the domain of transactional memory. Development in processor chips has shifted from providing increased clock speeds to focusing on multicore architectures. Traditional lock based approaches to concurrency control do not scale well and suffer from being complex and error-prone. An alternative is software transactional memory (STM), where synchronisation is achieved through in-memory transactions [ST97]. STM is an optimistic approach where threads modify shared memory without requiring prior synchronisation. After a thread has finished reading and writing, it validates against other concurrently running transactions to determine if a conflict has occurred. The result of transaction is to either commit the changes made or to abort, whereby all changes are rolled back and the transaction attempts execution again or terminates. The role of a contention manager in STM is to decide what to do in the event that two transactions conflict when accessing the same shared memory location, realised as a range of different policies. Evaluating the performance of a contention management policy is taken as a measure of throughput; the number of transactions that successfully commit in a given time period. A contention manager must also ensure that the properties of liveness and fairness are maintained such that threads can progress.

2.5. Dynamic Data Access

Data access patterns of a distributed application generates contention for resources. The previous section discussed how, through the use of a contention manager, this contention can be addressed in a manner that is both fair to the request owners but also can improve overall throughput. Another important aspect to consider for user's requests is the access relationship between a set of requests. The execution of this set of requests will invariably reflect the user's intent, based on the requirements of the request owner. Given the application type, managing this causality may be an additional requirement so that the user experience is consistent with the expected outcome.

Access Relationships

As discussed above, data access is the relationship between a set of operations such that the result of the first operation has been the catalyst for subsequent operations. Consider a simply online purchasing scenario. A user makes a purchase for a bicycle and in addition also buys some accessories to go with the bike; a helmet, set of lights and some mudguards. If each purchase is considered as a separate operation and the order is maintained as per how the user made the purchases then the purchase for the bicycle is the originating operation of the set and has the greatest level of impact. That is, the subsequent accessory purchases are made on the assumption that the bicycle has been successfully purchased. As such, the intent of the user is preserved in its entirety if all operations are successful. Similarly, if none of the operations are successful then the intent the user is not maintained. In between these two ends of the intent spectrum there is a another possibility to consider. If the bike purchase fails, then the user is unlikely to want a set of accessories and no bike to use them with. Ensuring that the purchase of the bike is successful has the highest level of importance in this set of operations. A weakening of the intent can be tolerated should the purchase of some or all accessories be unsuccessful. In this scenario, we have a spectrum of intent satisfaction where, depending on the operation type, certain operations must succeed while others can be tolerated to a greater degree should they fail.

This data access relationship property is also subject to change over time given a dynamic workload or application type. There will certainly be scenarios where the requirements of access are well defined and do not change but a dynamic approach,

similar to the idea of polymorphic contention managers, allows for changes to be made on the fly when this access relationship requirement changes.

Applying this idea of application level access preservation to a distributed application sees a shift from a scalable system that can tolerate inconsistent and conflicts to a more transaction-centric design. Maintaining the accesses between a set of operations is suited to the structure of a transaction; a set of operations where all changes are made else the transaction rolls back. Striking a balance between access preservation, data consistency and scalability is a requirement for a number of different application types in the distributed domain. In this thesis, three distributed application types are considered:

- **E-commerce**: This scenario is the setting for the bicycle example described previously. Client access is specific to each user operating in the system and user requests are competing such that each user wants their causality to be preserved over another. Client access is well defined in this scenario as there exists purchasing habits and products that are often bought together. As such, the access changes are often static or change slowly over time.

- **Massively Multiplayer Online Games**: Network gaming sees real time interaction between a number of collaborating users inside a distributed virtual environment. The access requirement in this setting sees an individual access for a given player between their own actions but also these actions have a causal impact of other players and the virtual environment. Actions and changes made to the game world happen within localised areas and affect players within a determined range. Accesses can be managed per region, termed a sphere of influence, where the impact of actions does not affect players contained within another sphere of influence. The accesses between player actions are well understood in this scenario, given a set of defined rules that govern the virtual environment and what the player can do. Requests arrive quickly, given the real time requirements of managing a virtual environment and require the ability to adapt to sudden changes in semantic relationships if required.

- **High Frequency Trading**: This final scenario sees rapid, quicker than real time, trading of resources (e.g. stock) where the access is relevant between the individual traders requests and also between traders. In these scenarios there is not scope to

roll back unsuccessful requests and as such these are simply considered unsuccessful trades in the system.

Predictability

As described above, the semantic relationship requirement has a detrimental impact on the scalability of a system that requires its maintenance due to the need for transactional like properties. However, this requirement directly lends itself to the notion of predictability. The actions of a user or client form a chain of requests where the next action that is part of this chain is predictable. These actions may be operations that are part of an automated system such as workflow, or may simply be issued as and when required by a user. In the case of automated requests there is a high level of predictability given application design knowledge. While there may exist branches, where different choices may be made, this structure is more formally defined than the requests issued by a user. In either case, this predictability, in combination with the access requirement, can be applied to the three case studies mentioned above. Each require a level of access relationship to be maintained and by utilising the predictability evident in each scenario this can be achieved.

2.6. Related Work

This section will survey related work in the areas of optimistic replication, eventually consistent systems and contention management policies seen in transactional memory.

2.6.1. Coda

The Coda File System [SKK+90, KS92] is an example of a distributed file system that implements many of the techniques of optimistic replication described in section 2.3. Coda's main aim is to provide constant data availability given the possibility of errors in the system elsewhere or when connectivity is transient or not available. Coda achieves this through the use of server replication and disconnected operations. Replication techniques allow for a greater level of availability and to tolerate partial failure in individual replicas while disconnected operations allow for a user to make progress while disconnected and, at a later date, integrate these updates into the wider system.

A Coda client may be in one of three states:

- **Hoarding**: This is the preferred and typical operating state where a client is connected to the network and relies on server replication. The state is named as such because the client, in anticipation of a possible disconnection, needs to hoard data. A user may have indicated that a set of files in critical while operating on a different set of files. Coda must ensure that both sets are cache on the client to ensure that operation can continue given the user requirements in the event of a disconnection.

- **Emulation**: The emulation state is entered when a client is no longer connected the network. As such, a Coda client must now emulate the role of the server allowing the user to continue progressing even when disconnected.

- **Reintegration**: When connectivity is reestablished to the network, the client must integrate changes made locally to the server while update the local cache to reflect server state. After reintegration is complete, the client enters the hoarding phase again.

Disconnected operation is transparent to user unless a local cache miss occurs or, during reintegration, a conflict is detected. A cache miss occurs when the required file was not retrieved during the hoarding phase. To avoid cache misses, Coda ensures files in the user's preference list are always up to date in the local cache. To combat the problem that it is difficult to always predict required files, Coda allows a user to provide a set of high level actions, and for the system to know what files these actions reference, to also maintain in the local cache.

Reconciliation in Coda is performed using a log based replay mechanism. During emulation, all operations are added to a log including the version state of the object that the operation was made against. At reintegration, the replay algorithm consists of three phases. Phase 1 parses the log and locks all objects that are referenced. Phase 2 sees each operation in the log is validated (to determine if there are any conflicts) and executed. Finally in phase 3 the changes are committed to the server.

In the event of a conflict, Coda first attempts to resolve it automatically. Automatic resolution in Coda relies on application semantics relating to directories whereby three conflict types exist that are not suitable for automatic resolution: *update/update*, *remove/update*, *name/name*. An *update/update* is a concurrent modification for the

same file, *remove/update* sees a file removed in one partition while being updated in another, and *name/name* sees two files with the same name created in two different partitioned replicas. Other directory operations can be automatically resolved using compensation in the form of create or delete operations. For conflicts that cannot be resolved automatically, a repair tool allows a user to manually resolve conflicts by overwriting inconsistent files or performing directory operations.

2.6.2. Bayou

With a similar goal to Coda, Bayou [TTP+95, PST+97] provides an optimistic replication scheme where a database can be replicated to a mobile device, modified while disconnected and then synchronised with any database replica that can be reached.

The update propagation mechanism seen in Bayou is called anti-entropy, using operation propagation over a state based approach, allowing for two replicas to bring each other up-to-date. A Bayou server consists of an ordered log of updates and a database which is the result of the ordered execution of the log. A second log, the write-log, contains all writes that have been received from this server or from other servers. When two servers communicate they need to reach an agreement on the writes stored in their two logs. A Bayou write operation consists of three components: a set of updates, a dependency check and a merge procedure. The dependency check and merge procedure are required components for conflict resolution. Anti-entropy sees two servers receives writes that they have not seen yet bringing each replica closer towards consistency (an eventually consistent system). As servers communicate and exchange updates, these spread epidemically throughout the system. Operation ordering uses a version vector and *happens-before* mechanism that provides a total order over writes accepted by a server and a partial order of writes in the system.

Conflict detection and resolution in Bayou is semantically based, using dependency checks and merge procedures respectively. Each write in Bayou contains a dependency check which describes the required query and the expected outcome of that query, given the state of the local replica it was applied to. When a write is transferred during anti-entropy, the dependency check is run against the receiving server's state. If the result does not match the expected result in the dependency check then a conflict is detected. At this point the merge procedure is executed in an attempt to resolve the conflict automatically. A merge procedure is essentially a programmer defined alternative

execution that still satisfies the requirements of the user. This procedure is executed against the current state of the replica at which the conflict was detected. Should the merge procedure also fail to execute then manual reconciliation must be applied.

2.6.3. IceCube

The IceCube system [KRSD01, PSM03] presents a technique for log-based reconciliation that considers the ordering of actions, applied at individual replicas, as an important factor for reducing irreconcilable conflicts. The reconciliation of logs is a combinatorial problem with the search space being significantly reduced through exploiting application level semantics. These semantics express whether an ordering of two actions is safe or unsafe. The outcome of reconciliation is a schedule containing an ordering of actions, all considered safe, that is used to reconcile the state of the divergent replicas.

Given a set of logs, each containing an ordered list of actions as applied at a replica, an optimal (i.e. has the fewest irreconcilable conflicts) schedule can be produced. This schedule is then used to reconcile the divergent replicas state and execution continues from there.

The IceCube system comprises of two distinct phases:

- Isolated execution phase
- Reconciliation phase, with three subphases:
  - Scheduling phase
  - Simulation phase
  - Selection phase

A replica is either in the isolated execution phase or in the reconciliation phase. When in the isolated phase, all actions are performed against the local shared objects. These actions move the replica state from an initial consistent view to a final tentative state. All the actions performed are recorded to form a log.

The reconciliation phase sees the logs of two or more replicas merged to generate a final schedule, an ordering on all actions, that can be used to reconcile the involved replicas to a mutually consistent state. The scheduling phase considers all possible combinations of actions to generate a number of potentially viable schedules that satisfy all static constraints.

Constraints represent the semantics of the application and determine whether the ordering of two actions is safe or unsafe. Given two actions α followed by β, their ordering is safe if their target objects differ or if α appears before β in the same log. If these two criteria do not apply then a further comparison must be made, and it is at this point that the application semantics are used. For this second ordering comparison, the ordering is provably safe (e.g. α and β are read primitives) or the application semantics consider such an ordering as safe. A result of unsafe indicates that the ordering of α before β is either provably unsafe (e.g. α is a write and β is a read for the same object) or the application semantics consider such an ordering as unsafe. A constraint may also return a third result type of maybe. A result of maybe indicates that the ordering of the two actions may be possible assuming their are no dynamic constraint conflicts. The scheduling phase produces a potential schedule that satisfies all static constraints but may contain some dynamic constraints that must be checked during the simulation phase.

The simulation phase takes the produced schedules and executes them against fresh replicas of the shared objects. A schedule that does not satisfy all dynamic constraints is aborted. A dynamic constraint can only be checked during the simulation phase because the semantics of the action depend of the current state of the target object. An example of a dynamic constraint is a bank account where a balance can either be credited or debited. Two credit actions commute and two debit actions commute subject to the dynamic constraint that the balance does not become negative. The simulation of a potential schedule is performed in steps with each step consisting of a prefix of successful actions (an ordering of actions that satisfy both static and dynamic constraints), the current simulated state of the objects and the next action to simulate. For this next action to be appended to the prefix schedule the dynamic constraint must evaluate as true and the action must successful execute against a shadow copy of the state. This new schedule and object state form the inputs to the next simulation step. If this simulation step fails then this simulation branch is aborted.

The selection phase takes the successful schedules of the simulation phase, ranks them and finally chooses the optimal schedule with which to reconcile the replicas. After selection, the reconciliation phase is considered complete and all replicas continue execution in the isolated phase.

The original IceCube system was extended, as described in [PSM03], to include a number of additional scheduling constraints to further express application semantics. Log constraints are a type of static constraint between actions within the same log. These allow the user and application to make their intents explicit. Preguiça et al. identify three log-constraint functions as being useful:

- **predSucc(α, β)**: Action β must execute only after α has succeeded. This provides a causal ordering between any two actions in the same log.
- **parcel**: This is a group of actions where either all execute successfully or none execute at all. While similar to transactional guarantees, a parcel does not ensure isolation, as the actions may be run in any order or interleaved with other non-conflicting actions.
- **alternative(α, β)**: This indicates that action β is a viable alternative should action α be irreconcilable. However, both actions should not appear in the same log. An example of an alternative might be two time slots for a meeting for a calendar application.

2.6.4. Cassandra

Cassandra [LM10] is a distributed NoSQL database system that, using replication techniques, provides a highly available and scalable solution to data management with no single point of failure. A Cassandra system makes copies of data, called replicas, and distributes these across available nodes as part of a cluster. The total number of replicas is referred to as the replication factor. The replication factor determines the number of replicas that exist; for example, if the factor is 2 then two copies of the data exist with each copy situated on a different node. These values will be determined based on network topology and the required level of availability and fault tolerance versus cost to manage.

Cassandra extends the notion of eventual consistency by providing tunable consistency where, for any read or write, the client application determines the consistency requirement based on requirements of response time versus data accuracy.

A write request is sent to all replicas while the consistency level determines how many replicas must respond in order for the write to be considered successful. The following are some of the possible write consistency levels:

- **ANY** : Provides the lowest level of consistency, the write must be made to at least one replica.
- **QUORUM**: A quorum is defined as: (replication factor / 2) + 1. For example, a replication factor of 6 provides a quorum of 4, tolerating 2 replicas being unavailable. A write must be made to the commit log and memory table of a quorum of replicas for this level.
- **ALL**: A write must be made to the commit log and memory table of all replica nodes.

Two types of read requests exist: a direct read request and a read repair request. A direct read request requires replicas to reply as determined by the consistency level set. A read repair is sent to all replicas that did not receive receive the direct request. A read repair may be required when the replication factor is greater than the consistency level set by the client request. This repair operation ensures consistency among all replicas. The following are some of the possible read consistency levels:

- **ONE**: A response is retrieved from the closest replica.
- **QUORUM**: Returns a record with the most recent timestamp from a quorum of replicas.
- **ALL**: Returns a record with the most recent timestamp from all replicas. If the read operation fails if at least one replica does not respond.

2.6.5. Contention Management Policies

The decision on how to handle a conflict between two transactions is implemented as a contention management policy. The approaches of these policies are described below:

- **Aggressive**: The manager always aborts the enemy transaction when a conflict is detected.
- **Polite**: This policy has the transaction that detects the conflict defer to the competing transaction by using exponential backoff. The transaction will wait for 2^{n} nanoseconds, where n is the number of retries while the conflict still exists. After a determined maximum number of retries has been reached, the enemy transaction is aborted (as in the aggressive policy).

- **Randomized**: When a conflict is detected, the enemy transaction is aborted or the original transaction pauses for random time interval up to a certain length. The probability of the choice and the maximum waiting time are adjustable parameters in this policy.

- **Karma**: This policy judges the amount of work a transaction has done so far when deciding which transaction to abort. Aborting a transaction that has just started is preferable to another transaction that is nearing completion. This is implemented by comparing transaction's priority. When a transaction commits, the thread to which it belongs has its priority reset to zero. After every successive object access, the priority for that thread is increased by one. When a conflict is detected, the priorities are compared and the enemy transaction is aborted if the current thread's priority is greater. If the priority is not exceeded, the original transaction backs off for a fixed period of time. If the enemy transaction is aborted it retains its current priority, providing a better chance of success in subsequent retries. This also allows for shorter transactions (those that do not access as many objects) to compete with much longer transactions.

- **Eruption**: Similar to Karma, this policy uses a priority mechanism that, when a transaction is blocked by another transaction with greater priority, the priority of the greater transaction is increased further. The rationale for this policy is to promote transactions that are blocking on objects critical to other transactions' progression and have the transaction finish quicker.

- **Kindergarten**: Transactions are encouraged to take turns accessing a block under this policy. For every transaction T, the contention manager maintains a list of enemy transactions that have caused T to abort. When a conflict is detected, if the enemy transaction is in T's list then the enemy is aborted. Otherwise, the enemy is added to T's list and T is backed off. If after a fixed number of backoff intervals T is still waiting for the same enemy to finish, T is aborted. When the thread retries T, and the enemy is still executing, the enemy will be found in T's list and be aborted.

- **Timestamp**: The aim of this policy is to provide the greatest level of fairness. For every transaction T, the start time is recorded. When a conflict is detected the start times of T and the enemy are compared. If T's timestamp is earlier, the

enemy is aborted. Otherwise, T is backed off for a series of fixed intervals. After half the maximum number of intervals, the enemy transaction is flagged as potentially being stalled. Once the maximum number of intervals has been reached and the flag is still set, the enemy is aborted. If the flag has been reset, due to the transaction performing execution, then T will double the wait interval and start the process fresh. Fairness is preserved by allowing a transaction to complete regardless of how slowly it runs or how much work it must perform.

- **Polka**: Polka is the result of combining the exponential backoff of the Polite policy with the priority accumulation technique of Karma. The result has a transaction back off for a number of intervals equal to the different between priorities of the enemy transaction. Polka also unconditionally aborts a group of reading transactions that hold an object required for a read-write transaction.

These range of policies were introduced in [SS04] where performance evaluation found that different policies work better for different benchmarks and that no single policy could provide all-around best results. Further work in [SS05b] identified the Polite, Karma, Eruption, Kindergarten and Timestamp policies as the overall better performers, and also introduced the Polka policy. The performance results identify Polka as providing the best overall performance across a range of benchmarks. In [GHP05] the notion of polymorphic contention management is introduced. While the performance results of Polka were favourable in a wide range of benchmarks, the notion of a universally accepted scheme does not favourably apply to a single application where concurrency patterns change dynamically. A polymorphic scheme aims to address this issue by deploying different policies depending on the dynamic requirements of the application. As an example, consider client requests over the Internet. During a period of high contention where many clients are submitting requests, one policy can be used. When the number of clients decreases, the system can switch to a second policy more suited to this type of workload. No one contention management or backoff scheme can satisfy all contention requirements for an application. While particular policies provide favourable results across benchmarks, the notion of a dynamic scheme that can apply contention management policies on the fly appears to be more favourable to address issues of unpredictable workloads.

Very recent work on transactional memory has seen an increased importance in

contention management as researchers attempt to attain a Universal Construction [Her91]. A Universal Construction is a procedure to transform any deterministic sequential object into a wait-free linearizable version. This object can then be updated concurrently by any number of threads. A wait-free implementation of a concurrent data object is desired as it guarantees that any accessing process will complete its operation in a finite number of steps. In essence, this may reflect a general purpose approach to a non-blocking contention manager with wait-free being the ultimate aim.

One may view this recent situation in the research related to Universal Construction in transactional memory as akin to the more traditional optimistic schemes in earlier work: there is a period of inconsistency whereby reads/writes/transactions are reordered to attain the highest throughput of commits. In the same way the earlier optimistic replication schemes identified application semantics as influencing throughput, the contention manager attains this status in transactional memory (i.e., the transactional manager is application dependent).

Therefore, if one could actually separate the contention management issue from the replication protocol one may introduce a variety of application dependent schemes for contention while utilising the same optimistic replication protocol. This is taken further by also generalising the contention manager itself using a probabilistic model of data access. As such, the primary concern of this thesis is contention management, attempting to first disconnect its implementation and then increasing its general purpose.

2.7. Gap Analysis

The background work presented in the section describes the key areas associated with the contribution of this thesis: contention management and replicated systems. A number of existing systems in the literature have been described in the context of their operation and contribution. Table 2.1 highlights the key areas of interest for the described systems.

2.7.1. Application Semantics

Coda, Bayou and IceCube all make use of application semantics in their operation. Their aim here is, through the use of specific application knowledge, to better handle any

System	Application Semantics	Reconciliation Scheme	Irreconcilable Conflict Management	Architecture	Consistency Level	Contention Management
Coda	Based on directory / file structure	Determined by operation semantics, log based replay	Manual reconciliation required	Client - Server	Eventual	Not used
Bayou	Dependency checks for writes	Anti-entropy protocol - replicas update each other	Manual reconcilation required	Peer to Peer	Eventual	Not used
IceCube	Programmer defined	Optimal ordering of updates	Manual reconcilation required	Client - Server	Eventual	Not used
Cassandra	Not used	Updates propogated in background to replicas, client defined	Not applicable	Replicated database	Client defined	Not used
Contention Management Policies	Not used	Transactions aborted and must retry	Transaction retries	Single machine	Strong	Policy based

Table 2.1.: Comparison of background system features

conflicts that may arise from scenarios where clients / replicas have operated with inconsistent data. Coda has well defined application semantics that are comprehensively understood within the computing domain. Using these, Coda based systems can allow clients to tolerate periods of disconnection with minimal impact for the end user. Bayou's use of application semantics extends further into the application domain by providing bespoke procedures to be executed in the event of a conflict. IceCube uses application semantics to inform its scheduling algorithm used during the reconciliation phase. In doing so an optimal schedule can be produced given a set of input operations. Cassandra does not make use of application semantics; the two modes of operation see a client making a read of write request to the database. Transactional memory, from which these contention management policies have been employed, also only considers the read and writes operations a transaction has performed as part of a concurrency control scheme.

Application semantics can be used effectively in replicated systems to better understand the context of operations performed at remote replicas. With this understanding, a decision can be made to whether or not operations truly commit. A simple example of this could be a room booking system. Without any application knowledge, two requests for the same room may conflict and only one can succeed. Extend this scenario by applying domain knowledge and it can be determined if their is a true conflict. Coda, Bayou and IceCube extend this to consider that, after determining if there is an actual conflict (by applying application semantics), there may be alternative executions that satisfy the end user's intention. In the room booking system, an end user may supply an alternative room that would be suitable avoiding a conflict.

While application semantics can be used successfully to fully understand the context of operations, they impose a large overhead for system designers and end users. In the Bayou and IceCube systems, significant work is required by application programmers to anticipate potential conflicts that could occur and develop suitable programmable solutions for these. By extending this to the end user and allowing for alternative executions to be provided presents an equal burden on system design and mode of operation.

Cassandra and transactional memory do not use application semantics in their operation. Application semantics could be used of part of an application that uses a Cassandra database-tier providing scalability and availability requirements. The same

can be provided for an application using a transactional memory approach for concurrency control.

2.7.2. Reconciliation Scheme

Reconciliation schemes are varied across the applications but share the common goal of detecting conflicts, applying application semantics (if applicable) and identifying irreconcilable conflicts. Coda, Bayou and IceCube all use log based reconciliation to update master replicas. In Coda, this log is generated when a client is operating disconnected from the network. This log is then played out against the master replica when a connection is made and conflicts are detected at this point. Bayou's architecture facilitates a different, but similar, process of exchanging logs through an epidemic propagation method. Finally, IceCube views the operation log as a scheduling problem; what is the optimal ordering of all operations such that there is the fewest conflicts.

Cassandra's approach to reconciliation of replicas is an interesting approach of client defined consistency. As such, data is updated across replicas in response to client requirements. This approach is used to afford the greatest level of scalability and availability at the cost of variable data consistency. Transactional memory does not have to contend with the potential for disconnected modes of operation as seen in Coda. As such, should a transaction be aborted, it can be resubmitted easily.

The potential for a client to operate against its local replica while disconnected produces the biggest challenge for reconciliation. As this period of time extends, the probability that updates made locally will be invalid increases. When considering large scale distributed systems where many updates are made every second, the requirement of disconnected execution becomes unfeasible should any execution or progress still be relevant given the overall context of the application state. The types of applications that Coda, Bayou and IceCube are aimed at do not have this high level of throughput such that periods of network disconnection can be tolerated. The concept of local replication and caching are still applicable for high throughput distributed systems and are employed to reduce network latency and server overhead where possible. As such, ensuring these local replicas and caches are up to date is still a relevant challenge for new systems.

2.7.3. Irreconcilable Conflict Management

The potential for irreconcilable conflicts within distributed systems is one that cannot be afforded. Application semantics and system design can be employed to reduce the number of conflicts and decide whether they are true conflicts. Coda, Bayou and IceCube all fall back on manual reconciliation to resolve such conflicts. The designers of these systems admit that designing mechanisms, such as the merge procedures in Bayou or the alternative function in IceCube, is a significant challenge. Within transactional memory, it is simple for any aborted transactions to be resubmitted and retried using new values for those data values accessed. When considering scalability requirements that high throughput systems require, disconnected modes of operation must be forgone to ensure that any operations that do conflict can be resubmitted.

An interesting area to consider if that of real time simulation. Such systems often employ local client caching of data, where execution is performed locally and sent to the server for verification. Due to communication latencies, in the event that a conflict is detected, it may already be too late to perform any reconciliation. This kind of scenario is seen within the massively multiplayer online games domain whereby a server must validate actions so as to update other clients within the region (other players) and to validate the input (the action performed is a valid one). With this in mind, it is desirable to reduce the number of conflicting operations to improve the overall game experience for the players.

2.7.4. Architecture

A range of architectures are used across the described applications. The client - server architecture presents a typical approach for the server and database tier managing the master copy of the replica with clients managing their own subset of required data locally. Communication between the two is used to present updates and to receive updates to the local state. Bayou's approach is to use a peer to peer propagation method allowing two communicating replicas to update each. As replicas encounter each other, updates will eventually be applied at all replicas. The peer to peer approach presents all the challenges a regular peer to peer system presents, such as group membership and peer discovery. Such systems typically employ a centralised server to manage these requirements. Again, when considering high throughput systems, a peer to peer approach provides many disadvantages and when requiring a centralised server for

management we move back towards the usual client - server architecture.

Cassandra is not a complete system solution but rather a database tier that can be employed with a desired architecture. As such, application semantics could be used at an application layer to further enhance the system. Transactional memory resides at a similar level and is rather a component of a larger system.

2.7.5. Consistency

Among optimistically replicated systems a form of weak consistency termed eventual consistency is employed. This is due to the nature of how the replication scheme facilitates clients to work independently and update as when possible. The benefits and tradeoffs of strong versus weakly consistent systems have been explored in this chapter. While strong consistency provides guarantees for data access, it does not provide the scalability and availability requirements that are demanded by modern distributed systems. However, when choosing a weakly consistent system, the additional challenges of conflicts and ordering must be addressed.

2.7.6. Contention Management

Contention management has seen success in many systems and more recently within transactional memory. Existing optimistically replicated systems have not considered the challenge of managing conflicting operations as a problem of contention. Coda, Bayou and IceCube aim to leverage the application semantics to address conflicts or better order updates such as to reduce the overhead of conflicts. However, when considering a contention manager, the function is similar to that of a scheduling. As access to data is mediated by the contention manager and the policy it employs, a natural schedule is created among the competing processes. A contention manager can also address the demands of high throughput experienced in distributed systems through this access mediation.

2.8. Discussion

Contention management has been used in a range of domains to mediate access to a group of resources. As processes compete to communicate over Ethernet, a contention manager is employed to schedule access to the medium. In transactional memory, a

number of processes compete for access to a shared data set. The contention manager here balances access to shared data using a contention management policy as specified by the application developer. As described, a number of different policies exist with some providing more favourable performance (in terms of throughput) and fairer access. Without a contention manager in these scenarios, a process will attempt to access the resource as fast is permitted, be that clock cycles of the CPU or latency introduced by the communication medium. No benefit is achieved in this approach; communication packets collide in Ethernet and transactions conflict with each and abort in transactional memory.

As such, the requirements of a contention manager are to:

- **Manage access to shared resources**: the contention manager employs a policy in which access is scheduled. This is typically achieved by making a conflicting process wait a period of time before attempting access again; this is termed as a backoff period. Exponential backoff has been shown to perform well in Ethernet while a number of different policies have been explored in the transactional memory domain.
- **Reduce the number of conflicts**: a conflict occurs when competing processes attempt to access the shared resource at the same time. A contention manager's goal is therefore to avoid this scenario and as such reduce the overall number of conflicts that occur.
- **Increase access throughput**: without a contention manager, processes attempt to access the resource as quickly as possible. Therefore, throughput is measured as the level of successful accesses to the resource. A contention manager should improve the throughput in comparison to that of no contention management.

When investigating optimistically replicated systems, these type of applications operate with similar requirements to that of a contention manager. Clients, with a copy of some shared data, compete with each other to make updates to the shared data in the context of the application. The owner of the master data set (typically a server receiving these update requests) must determine which requests can be satisfied and which conflict. The focus of this work has been how best to use semantic information at the application layer to determine whether a conflict can be avoided by some alternative execution of ordering.

These systems have not viewed the problem of managing conflicting access as an issue of contention. As one of the requirements of a contention manager is to reduce the number of conflicting accesses, it would seem apt that such a technique could be applied to the domain of distributed systems and optimistic replication. It is also interesting to note that contention management schemes have equally not considered how the semantic information of the application can be used to inform the contention manager. This kind of information can be used to identify sets of data that is being used more often and adjust the contention management policy dynamically to adapt to trends in the access.

This thesis aims to bridge this gap by exploring the use of a contention manager in a distributed application. The use of semantic information about the application has already been shown to successful reduce the overhead of conflicting operations and is introduced as a key part of the policy employed by the contention manager. The aim here is to be able to adjust to changing access patterns an application may exhibit during its execution. Naturally, some resources may vary between low and high contention and a manger must be able to address this. As mentioned, contention managers have also not used application knowledge to support a policy and so this work aims to show that both domains can benefit from these techniques.

3. Design and Implementation

The background work of this thesis identified an avenue of research into the use of a contention manager for distributed applications. Contention managers have been deployed in other domains to afford a greater level of throughput and reduce the overall number of conflicts between competing processes. The issue of conflicting accesses is one that has great importance to distributed systems that use a degree of replication. As such, the thesis aims to address these two areas by bringing contention management to the domain of distributed applications. This chapter presents an abstract model for a distributed application from which an experimental instantiation is produced and described in detail. This instantiation is used to create a test bed from which to gather statistics important to determining how successful the contention manager has been.

3.1. System Model

The model considers a distributed application as a finite set of processes that communicate with each other via message passing. A process may either be *available* or *unavailable* and can transition from either state. A process may become *unavailable* as a result of a crash, and return to *available* after a period of recovery. When *available* a process operates according to a defined algorithm as assigned and does not behave maliciously. Communication between two processes is assumed to be reliable; messages are not duplicated and a message sent from process p_i will eventually arrive at process p_j should both processes remain *available*.

Three process types exist in this model - clients, application servers and database servers. The following sections describe the operation of each of these process type.

3.1.1. Client Processes

Clients are modeled as processes and are denoted by c_1, c_2, ... c_i ($c_i \in Clients$). User to client interaction is not modeled in this system and client processes do not communicate with each other. A client process invokes the *clientUpdate()* function providing an update request as a parameter. This update request reflects a state change that has been applied to data held locally by the client. The result of the *clientUpdate()* function is the creation of a message m conveying the state change. This message is then delivered to an application server process. For every state change to local data, a client process performs the *clientUpdate()* function, generates a message and send it to an application server process. As such, a message m, sent from client process to application server process reflects a single state change. A message cannot contain multiple state changes; the *clientUpdate()* function is executed for each change and a new message is generated and sent. There is no expectation that a reply will be received in response to this message. If a message does arrive at a client process from an application server process then the *clientReceive()* function is performed. This message informs the client process that a previous message, sent to the application server process as a result of executing the *clientUpdate()* function, could not be successfully applied. The content of the message received from the application server process instructs the client process to roll back to the point of execution where the invalid state change was performed and attempt execution from this point.

3.1.2. Application Server Processes

Application servers are modeled as processes and are denoted by s_1, s_2, ... s_i ($s_i \in AppServers$). The role of an application server process is to receive messages from client processes, determine if the state request is valid and inform the client to roll back if it is not. A number of application server processes may exist; the number required is an issue of load balancing and would be addressed as part of an implementation of this model. Application server processes do not communicate with each other and all application server processes use the same algorithm for message processing and contention management. Once established, a client process will always communicate with the same application server process. When a message m (sent from a client process to the application server process) arrives, the *appReceive()* function is executed. This function executes as follows:

1. The application server process first must decide whether the message received should be processed any further. A client process, as viewed by the application server process, exists in one of two states at any given time - *active* or *passive*. An *active* client process indicates that this process is not currently a part of the contention management policy. A *passive* client process indicates that this process is currently part of the contention management policy and the message should not be processed any further; the *appReceive()* function terminates at this point.

2. The application server process must determine if this state change is valid in relation to what the current state of the data is as held by the database process. A message is sent from application server process to database server process requesting the current state of the data. On receipt of the reply message from the database server process, the application server process compares the received state update from the client process with the actual state provided by the database server process. If it is determined that the update made by the client is valid then the application server process creates new a message containing the new state as supplied by the client and sends this to the database server process to apply the change. No reply is expected back from the database server process and no message is required to be sent to the client process so the *appReceive()* terminates at this point.

3. Should it be determined that state update supplied by the client process is not valid then the contention management policy is enacted; the server process executes the *appContention()* function and the *appReceive()* function terminates at this point.

The *appContention()* function applies the contention management policy. The contention management algorithm employed uses a backoff approach; contention is encountered when the client process has failed to successfully update the data state as per the state held at the database server process, due to another client process updating the state. As such, this is where contention is present within the model: client processes competing to modify data state. A time period (this period may be measured using different metrics but wall clock time will be used in this model) is determined as per the policy and the client process is now in the *passive* state. When the time period for a client process that is in the *passive* state expires, the *appSend()* function is executed. The application server process generates a new message informing this client process to

roll back (as per the *clientReceive()* function).

3.1.3. Database Server Processes

Database servers are modeled as processes and are denoted by db_1, db_2, ... db_i ($db_i \in Databases$). The role of a database server process is to manage the master copy of the data that a client process makes state changes to locally. A number of database server processes may exist in the model; this requirement is one to addressed as per an instantiation of this model and the specific design constraints. A database server process accepts messages from application server processes; when a message arrives the *dbReceive()* function is executed. The database server process responds to the request of the application server with the current state of the data in question and the *dbReceive()* function terminates.

3.1.4. Model Discussion

This model provides an abstract description of the processes involved, their methods of communication and their operation. As per the gap analysis, the model is representative of the requirements identified:

Architecture

The focus of this work is looking at contention management in the context of distributed applications. This domain supports a large number of different architectures and designs to support varying application requirements. The decision was made to pursue the issues of contention management in the scope of a three tier architecture. This architecture represents a large domain of applications, both in the existing literature presented in the background section, and the case study domains to be explored. Both Coda and IceCube are designed around autonomous clients communicating with a central server that manages the state across all clients. While Bayou supports a peer to peer method of communication, client processes are still evident and it relies on a form of server to act as a master owner of the replicated data. Cassandra is not a complete system by itself but could be deployed as a database tier within a three tier application.

The case studies described within this thesis are natural candidates for a three tier architecture. The e-commerce scenario matches this architechture very closely, whereby clients issue purchases to a central server that determines, by querying the database

state, whether such purchases can be satisfied. The trading scenario is similar, but with different functional requirements on latency. The online gaming scenario sees client processes as players in the virtual world and this world being supported by a number of application servers that provide a governance over the actions of clients. The requirements here are ones of real time communication where locally replication is required to offset this overhead.

Contention Management

The applicability of contention management for distributed applications is the focus of this thesis; the existing work, in the domain of optimistic replication, has investigated the applicability of application semantics to reduce the overhead of conflicting operations. Application semantics are clearly beneficial to this cause and this thesis also considers how contention management, shown to reduce contention and improve throughput, can also support these application types. The decision to investigate contention management in the form of a back off algorithm was made given the proven success as per the background research. A number of the contention management policies in transactional memory also use the back off approach successfully.

Early research for this thesis considered the priority approach, similar to that of the eruption policy of transactional memory. In this scenario, if a client process conflicted when updating the data state then it was given priority for a subsequent access. This was enhanced using the application semantics and expected execution path of the clients to reserve future data such as a client was guaranteed to make progress. The results of this work proved to provide little to no impact on the level of conflicts clients were experiencing. It was found that the priority approach benefited from a single client process to access the required data and as such the overall conflicts were not impacted. Throughput was also not improved, as clients were being blocked access regardless of whether the data update was valid, due to the reservation mechanism.

Application Semantics

This has not been specifically addressed in the abstract model, to support a clear and concise presentation. Application semantics for optimistically replicated systems have been identified in previous research and in the gap analysis as critical to the reduction of conflicts. Coda, Bayou and IceCube all use operational semantics to reason about the

nature of conflicts and how best to manage them, whether this be operation ordering as in IceCube or possible alternative executions as in Bayou. This thesis set out with the question of how to best manage the intent an end user has with regards to the actions performed in a replicated system. As operations are submitted in isolation, assumptions are made as to their results that can eventually turn out to be invalid given the overall state of the system, as other users are also making changes. This requirement is reflected in the contention management described in the following section. Application semantics pertaining to intent and actions performed are described and reflected as a graph data structure that the contention management policy uses to determine suitable back off periods.

Conflict Management

What is common throughout the literature for optimistic replication is that managing conflicts is both application dependent and a formidable research challenge. Often the requirements of a user, as reflected in the state changes made, are the only suitable result. Alternative executions as in Bayou are a possible solution, but require the application semantics to support this (e.g. alternative room bookings in a calendar application) and are also very difficult to predict in advanced. This thesis has chosen to focus on a broad domain of distributed applications (as described in the case studies) and how best to mitigate conflicts at the point of execution rather than how best to design solutions to for alternative executions that are bespoke and application dependent. This is why client processes are require to roll back their execution rather than tolerate and repair using alternative execution. This also satisfies the intent requirement as the ordering of operations as submitted by the client is guaranteed. The option to support alternative execution as a means of repairing irreconcilable conflicts is not ruled out by making these design choices, it is felt that the first area to focus research is that of contention management and how the semantics of the application can support this.

Consistency

Optimistic replication supports eventually consistency and the model describe also provides this consistency level. If a client process never receives a message from an application server process then either: (i) all client process messages are honoured and states are mutually consistent; or (ii) all application server processes or client process

messages are lost. Therefore, as long as sufficient connectivity between client and application servers exists, the shared data will become eventually consistent. Such consistency levels support scalable distributed applications that experience high levels of throughput.

3.2. Application Semantics

As previously described, application semantics are a critical component in supporting conflict management. Existing systems use application semantics in different ways, but with the same overall goal of reducing the number of conflicts. Bayou uses semantics to apply alternative executions in the event of a conflict detection, while IceCube uses the semantics of operations to better order updates and achieve an optimal schedule with the least number of conflicts. This thesis has identified that the overall intent of a user is also an important factor when considering operations submitted and the potential for conflicts. There exists links between the execution of operations that, should one fail and the other succeed, the actual intent that the user of the system had may have been violated. Such examples can be seen in shopping basket transactions in e-commerce whereby conflicts may occur in stock levels but the user expects the complete order to be satisfied. Taking this notion forward, by identifying that contention management supports a reduction in conflicts, application semantics can also be used to augment the contention management policy.

The notion of a data item is used to abstract from implementation specific concerns of representation. An instantiation of this model may represent a data item as a file that has been updated (e.g. the Coda file system), a unit of application logic (e.g. purchasing an item from an online marketplace) or an SQL query to be processed at the application and database tier. This abstraction is important as it allows for a semantic model to built for the application domain that represents the intent of an end user. A number of data items forms a data set that represents that is shared between all processes in the model. The term access is used to indicate that a client process has used a single data item as the focus of an update and this is communicated to the application server process in a message.

To represent the application semantics, this data set is used to a create a graph $G = (V, A)$ where V(G) is a set of vertices where each vertex represents a single data item and

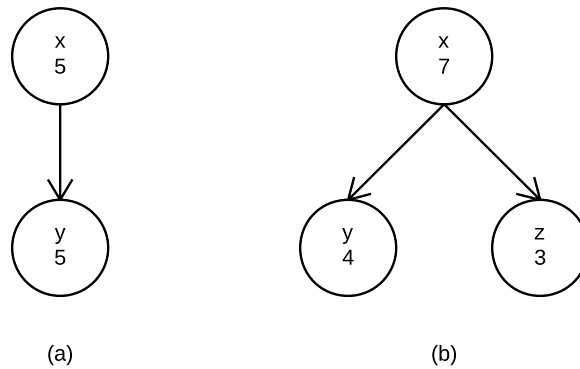

Figure 3.1.: Two basic graph configurations

A(G) is a set of ordered pairs of vertices called arcs. An arc $a = (x,\ y)$ is directed from vertex x to vertex y. A function f exists for the graph that provides a mapping $f\ :\ V(G) \rightarrow X$ where X is an integer that indicates the access frequency of the vertex. An arc $a = (x,\ y)$ identifies a semantic relation between these two data items x and y such that the following holds true: for any given client process who updates data item x, the focus of the next update will be data item y. Figure 3.1 shows two simple configurations of graph. In (a) there exists graph $G = (V,\ A)$ where $V(G) = \{\ x,\ y\ \}$, $A(G) = \{\ (x,\ y)\ \}$ and $f(x) = 5$, $f(y) = 5$. For a client process to access data item x, the graph in (a) identifies y as the focus of the client's next access. In (b) there exists graph $G = (V,\ A)$ where $V(G) = \{\ x,\ y,\ z\ \}$, $A(G) = \{\ (x,\ y),\ (x,\ y)\ \}$ and $f(x) = 7$, $f(y) = 4$, $f(z) = 3$. Having accessed x, the graph in (b) identifies data items y or z as the focus on the client's next access.

Thus far, the description of the graph structure identifies the relation, represented as arcs, that data items, represented as vertices, have to each other. This describes the application semantics as they relate to the intent of client processes in issuing updates. The graph structure is merely a representation of the data items held by the database; another representation could be chosen by this graph structure that suitably supports the requirements, while naturally supporting traversal as required by the contention management policy.

The contention management policy, employed by the application server process, uses this graph structure to manage the contention dynamically as it processes messages received from the client. The function f returns the value of the vertex: this represents the frequency with which this data item has been accessed. When an application server

process receives a message from a client that successful updates the data item, this frequency value is increased. This increment is a parameter of the contention manager and can be adjusted as per the requirements of the implementation. A large increment may be suitable where there are a large number of client processes due to the larger number of accesses. Over time, contentious data items will experience larger frequencies. It must be noted that these values cannot rise without suitable management. Two possible options are to apply a maximum value that these frequencies can reach before being reset or to determine when a data item is no longer contentious possibly by applying time periods within which an update must be seen. This management choice is one that would be implementation dependent and may be suitable in different configuration scenarios.

3.3. Implementation

The following section provides an experimental instantiation of the described model that is suitable for three tier application. This instantiation is used to generate the simulation test bed with which the contention management policy is evaluated.

3.4. System Model

The distributed application is deployed within a three-tier architecture as illustrated in figure 3.2. Communication between client, application server and database server processes takes the form of message passing with the requirement of reliable transmission (ordering and message loss handling). These can be addressed using a communication protocol such as TCP. Message loss is supported as per the implementation description to support other communication protocols that do not provide the guarantees of TCP.

Data accesses are performed on a local replication of the database state at each client. Deciding on a partial or complete replication of the database state for a client process is largely application specific. Considering the case studies presented in section 2.5, complete replication is required in the e-commerce and high frequency trading, while partial replication is suitable for online gaming. For e-commerce and high frequency, the actions performed by users could see products or stock purchased from all available listings. Techniques in online gaming such as sharding or areas of influence suit a partial replication scheme. Such techniques group users based on geographical region within the

virtual environment. The shared data and state is locally contained within that region and typically only sees users interacting with other users in the same region. As users move from one region to another, the old replica can be discarded to adopt a fresh replica pertaining to the shared state of the new location.

Clients inform the application server of any accesses performed locally. The designated application server, as managed by the load balancer, receives these access notifications and determines if these accesses are valid. If these updates are valid, the database state is updated to reflect the changes made locally by the client. However, if an update results in a conflict then the client is notified. When the client learns that a previous update was conflicting, that client will roll back to the point of execution where this action took place and resume execution from this point.

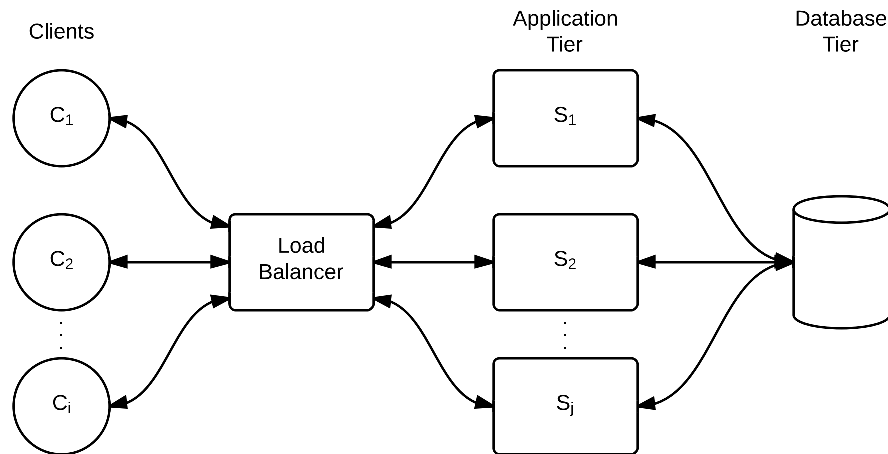

Figure 3.2.: System model

3.4.1. Load Balancer

The load balancer distributes the number of client processes across available application server processes. A sticky session is used to ensure that requests for a single client are always directed to the same server process. This session is maintained by the load balancer and when a message is received from a client process, the balancer will pass the message to the application server assigned. For new clients, the assignment of application servers is done through round robin such that there is an even load. Each server process maintains metadata on a per client basis pertaining to the contention manager implementation. Rather than require servers to communicate between each

other to share metadata, the design choice to use sticky sessions was made. The trade off of avoiding this communication overhead must be considered in the event of a single server process failing. Periodic writing of this metadata to stable storage would not be feasible as, when the server recovers, the client requests will be directed to a different server and the metadata the server has will now be out of date. As such, should a server fail, the loss of this metadata is tolerated given: a) the overhead of establishing a new session for a client with a different server and the generation of initial metadata is minimal, and b) the extra rollback required by an affected client due to the loss of metadata can be greatly mitigated by the load balancer in detecting the failure quickly and directing new requests to a stable server. Mechanisms for handling server failure and recovery are not a focus of this work, but are considered given the extensive scope of requirements when designing a distributed application.

3.4.2. Database

The database manages the master copy of the shared data set. The data set is modeled as a set of objects called data items, labeled $\mathrm{DI_1}$, $\mathrm{DI_2}$, $\ldots \mathrm{DI_k}$. Each data item has a logical clock value associated to it. The data item reflects the state of the object while the logical clock is used to record the version of the object. The logical clock value is requested by the application server to be used in determining whether a client's update message can be honoured or if it is in conflict. The database accepts requests to retrieve logical clock values for data items or to update the state and logical clock of a data item. An update of a data item occurs when a client message is honoured (the updated is not in conflict) by the application server. The interaction between application server and database occurs using transactions and is managed by the database manager.

3.4.3. Clients

Each client maintains a local replica of the data set as managed by the application servers and stored at the database tier. All client actions, as performed through interaction by the user with the application, that access shared data are enacted against the local replica initially. The client uses a number of logical clocks to aid in managing their execution and rollback:

- **Client Data Clock (CDC)**: For each data item there exists a CDC that identifies the version of the data item's state as contained in the client's replica.

This value is used by the application server to identify when a client's state of the data item is out of date. This value, represented as an Integer, is incremented by the client when an update is made to the state of the associated data item. It may also be modified, updating it to the most recent value, if a message is received from an application server informing the client of a conflicting access.

- **Client Session Clock (CSC)**: Represented as an Integer, each client maintains a single CSC. This value is attached to every update request sent to the application server. This value allows the application server to ignore messages belonging to out of date sessions. A session reflects a set of messages sent between one rollback and the next. When a client is requested to rollback the CSC is incremented by one. Given transmission delays and message queuing at the application server, it is possible that a client may perform updates in the period between an application server detecting a conflict and the client processing the associated rollback request. These updates should not be processed and the session clock lets the application identify these out of date requests.
- **Client Action Clock (CAC)**: This value, another Integer, is incremented each time a message is sent to an application server. This value is used by the application server to recognise any missing messages from clients.

The result of an action that modifies a data item in the local replicated state results in a message being sent to the application server. This message contains the new state of the data item the client has updated, the CDC for the data item, the CSC and CAC. A ordered execution log containing the content of each message is maintained by each client to allow the client to rollback.

A message arriving at a client from an application server indicates that a previous update was in conflict given the state of the data item at the server-side or that a prior message has gone missing. In either case, upon receiving a message from the server, the client compares its CSC with the session clock contained in the received message. If this clock is equal or lower to the client's CSC then this application server message is ignored. This is because the client has already rolled back; it is possible for the application server to send multiple copies of the same rollback message.

If the message was sent from the application server due to a missing client message then it will only contain an action clock. This message type is called a missed message request.

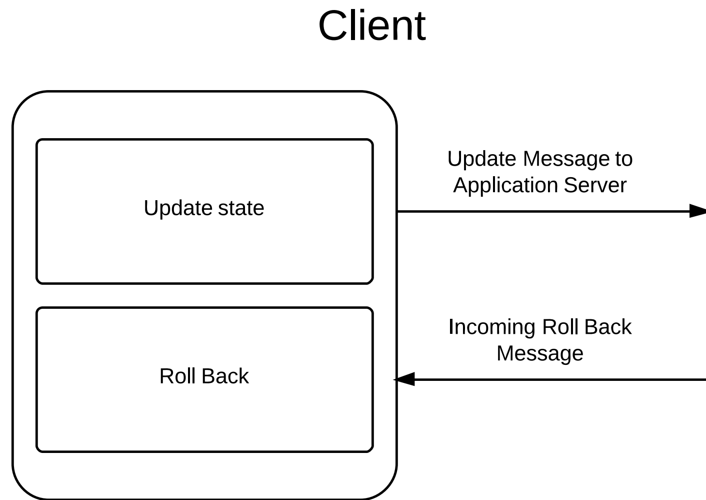

Figure 3.3.: Client Model

On receiving this type of message, the client should rollback to the action clock using their execution log. If the message was sent from the server due to a conflict then this message will also contain the latest state of the data item and logical clock value. This message type is called an irreconcilable message request. On receiving such a message, the client halts execution and rolls back to the action clock using their execution log.

Although a client will have to rollback when requested by the application server, the receiving of an application server message also informs the client that updates previous to the point of rollback were successful. As such, the client can reduce the size of their execution log to reflect this. Appropriate log truncation is also required in the event that no rollback requests are received from the application server for a long period to avoid this log growing very large. Should the server not generate a rollback message request for a client in a given period, a message will be generated and sent to the client informing of successful updates. This can be achieved by providing the client with the latest CAC value the server has committed; any updates prior to this were successful. The period in which these messages are sent to a client are specified given the deployment environment. A fixed period may be suitable or a dynamic period using the request throughput as a regulation metric.

Figures 3.3 and 3.4 illustrate the client model and workflow.

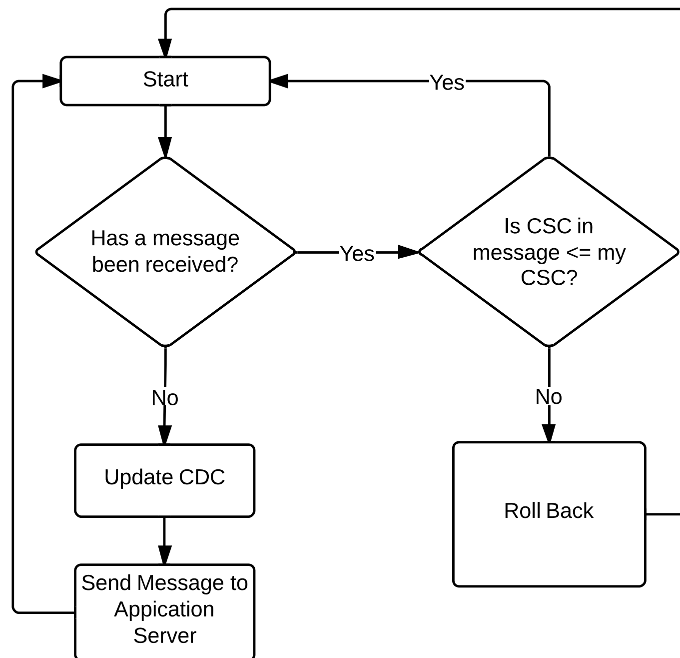

Figure 3.4.: Client Workflow

3.4.4. Application Server

The role of an application server is to manage the causal relationship between a client's actions and ensure a client's local replica is eventually consistent. The application server manages three types of clock to determine the validity of a client's update message:

- **Session Identifier (SI)**: This is the application server's view of a CSC; the server maintains an SI for each client it is managing, for a client $\mathrm{C_i}$, labeled as $\mathrm{SI_{C_i}}$. This value is used to disregard messages that belong to out of date session as described in section 3.4.3. When an application server requests the client to roll back this value is incremented by one.

- **Action Clock (AC)**: This is the application Vaserver's view of a CAC; the server maintains an AC for each client it is managing, for a client $\mathrm{C_i}$, labeled as $\mathrm{AC_{C_i}}$. This value is used to identify any missing messages from a client. Every update request from a client that is honoured by the server results in this value being set to the CAC as received from the client.

- **Logical Clock (LC)**: This value is stored with the data item state at the database tier and is requested when an application server performs a conflict check. The application server compares the CDC received with the true value from the database to determine if a client has operated on an out of date version of the

associated data item. If the update from the client was valid then the application server requests the database to update the state and value of the data item the client operated on.

A message received from a client, say C_1operating on DI_5, may not be able to be honoured by the application server due to one of the following situations:

- **Stale session**: The CSC_{C_1}in the received message is not equal to the application server's SI_{C_1}.
- **Lost message**: The CAC_{C_1}in the received message is two or more greater than the application server's AC_{C_1}.
- **Stale data**: The CDC_{C_1}in the received message is not equal to the application server's LC_{DI_5}as requested from the database.

When the application server has to rebut a client's request, a rollback message is sent to that client. Preparation of this rollback message depends on the state of the client as perceived by the application server. An application server can recognise a client, C_1, as being in one of two states:

- **Progress**: The last message received from the client was honoured.
- **Stalled**: The last message from the client could not be honoured or was ignored.

If the state of $C_1$is Progress then the application server will create a new rollback message and increment $SI_{C_1}$by one such that messages from this session are no longer processed. If the message could not be honoured due to a lost message then the application server creates a missed message request containing the current value of $AC_{C_1}$and sends this to C_1. This will allow the client to rollback to the last successful update in the execution log. If the problem occurred due to stale data (clock values are out) then the message sent to the client will also include the latest $LC_{DI_5}$and the latest state of DI_5as requested from the database. This message is the irreconcilable message request as described in section 3.4.3. The application server moves C_1 from the Progress state to the stalled state and records the rollback message sent, collectively called the authoritative rollback message.

If C_1 is already in the stalled state then any message arriving from $C_1$is replied to, by the application server, with the current authoritative rollback message. The exception to

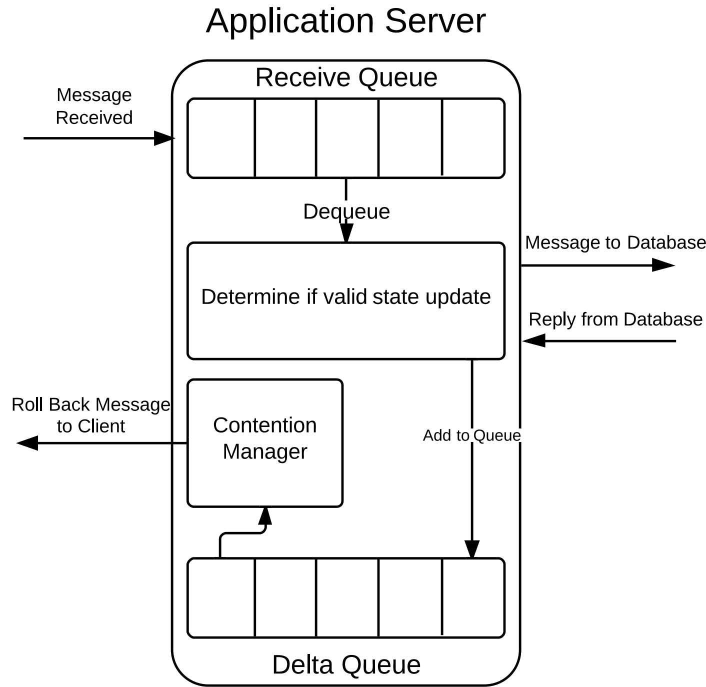

Figure 3.5.: Server Model

this is if the newly received message contains a CSC_{C_1} value equal to SI_{C_1}, indicating that an authoritative rollback message has been received by $C_1$and processed. However, the received message, while containing a valid session, may be received after a prior message has gone missing. As such, the CAC_{C_1} value is compared with the AC_{C_1} value by the application server. If the client's value is greater than the servers, indicating a message with the correct action clock has gone missing, then a new authoritative rollback message is created containing the current AC_{C_1} value and is sent to the client. The application server also increments SI_{C_1}. In the event that both the received CSC and CAC client are valid then the logical clock values are compared to determine if there is a conflict. If there is no conflict then the client's update is honoured and the changes are stored at the database, the client is moved to the Progress state and the authoritative rollback message for $C_1$is discarded. If the client is found to be in conflict then SI_{C_1} is incremented and a new authoritative rollback message is created and sent to C_1.

Figures 3.5 and 3.6 show the structure of the application server process and the work flow that follows when a new message is received. Note that server will always check for expired delta queue messages after processing a single message.

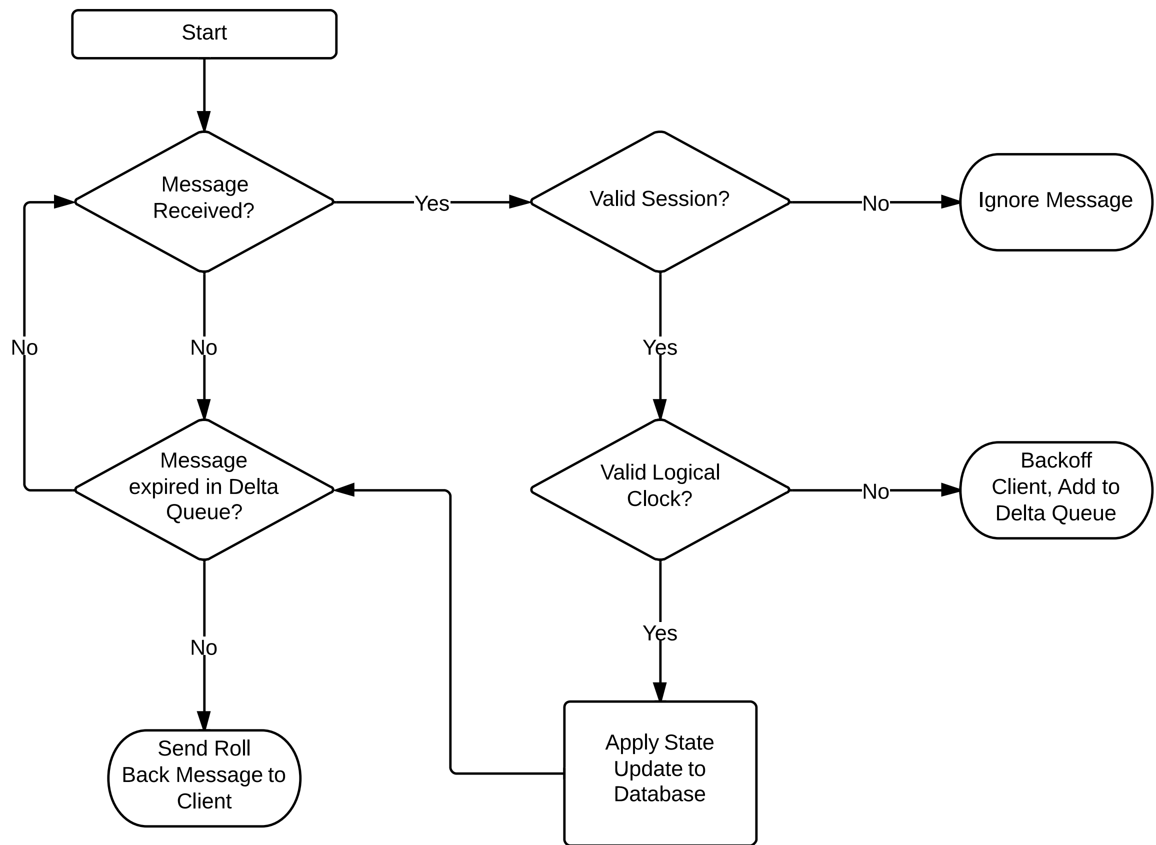

Figure 3.6.: Server Workflow

3.5. Semantic Contention Management

The proposed system model in section 3.4 presents a base model of an optimistic replication scheme. Client processes manage a local replication of shared data and communicate changes to this by sending update requests to the server. This section will now introduce the contention management framework as applied to the model described thus far. The aim of the contention manager is to reduce the number of conflicting actions that occur as a result of a client operating on out of date versions in their local replica. The contention manager should achieve this without hindering the overall throughput of successfully committed actions.

3.5.1. Contention Manager

Contention occurs when multiple clients attempt to access the same shared data item. This access is first made locally to the client's replica and is eventually propagated to the server. The first client message with the correct logical clock value to be processed will be successful in updating the state of the data item. All other clients who accessed the data item with the same or lower value of the clock will be conflicting as the clock will

have been increased after the successful access. It is at this point that the contention manager operates.

Each application server manages their own graph configuration representing the data items stored within the database. The result of this will be that graphs will diverge in context between different application servers. This is not a problem as the graphs represent the in-session causality of the clients the server is managing, not all clients as part of the whole system. Managing a single graph between all application servers would be a considerable overhead and would not scale with increasing numbers of clients and application servers. The base system described in the previous section is extended with the following constructs to support the contention manager:

- **Frequency (F)**: A value is associated with each vertex in the graph indicating the frequency of access for the given data item. The frequency of a vertex is increased when a client successfully updates the associated data item. Along with the data item that was the focus of the action, any neighbouring vertices (the target of any outgoing edges of the focus vertex) also have their frequencies incremented to model the expectation that one of these data items will the be the focus of client's next action. To ensure that these frequencies do not increase indefinably, the frequency value of each vertex is periodically decremented when the vertex has not seen any accesses since the last period. A ceiling value for the maximum that the frequency can reach is also needed such that popular vertices do not have their values steadily increase.

- **Delta queue (DQ)**: When a client's action could not be satisfied by the application server due to a conflict (out of date logical clock values), rather than inform the client straight away, a backoff period is generated. This period is generated based on the sum of frequencies of the associated vertices in the contention graph. The associated vertices include the data item for which the conflict occurred along with the data items with the highest frequency values as defined by the implementation configuration. This metric defines a number of hops to make in the graph. The larger the number of hops, the greater the value will be and therefore the longer the back off period. This value can be adjusted to suit the application domain. If any vertex has no outgoing edges then the backoff period is taken as the sum of the frequencies so far. The reasoning behind taking the

frequencies of the three most volatile vertices is based on the expectation that a client's next action will be for this data item, given that it is the most volatile of the set of outgoing vertices. A client, for which a backoff period is required, is now considered in the stalled state and is represented as an element of the delta queue. Client elements exist within the delta queue for the given backoff period, measured in milliseconds.

- **Enhanced authoritative rollback message**: When the backoff period for a client in the delta queue expires, an enhanced authoritative rollback message is generated and sent to the client. This is an extension of the authoritative rollback message described in the based model which includes a partial state update. This partial state update includes the latest state and logical clock values for the conflicted data item and data items that are causally related to the original conflicting access. Based on the assumption of causality as modeled in the graph the aim here is to preemptively update the client such that future update messages will have a higher chance of being valid.

3.5.2. Dynamic Reconfiguration

To tolerate dynamic changes to the access patterns over the life time of the distributed application a mechanism is required to modify the structure of the previously static graph. The static graph structure may be a suitable option for systems where the access patterns are well understood and are not subject to change. When considering this in the context of a long lived system then it is unlikely that the access patterns for users will always remain static and so the ability to identify and react to these changes in the structure of the graph is required.

To satisfy dynamic reconfiguration of the graph, two new additions are required at the application server:

- **Happens Before Value (HBV)**: The vertex representing a data item a client last successfully updated.
- **Happens After Value (HAV)**: The vertex representing a data item a client successfully update directly after the HBV.

If no edge exists between the HBV and the HAV then one is created. If this process was to continue in this manner then the result would be a fully connected graph which would

not be representative of the causality. Therefore, there needs to be an additional mechanism to record how often an edge has been used (how often there has been a causal link between two data items):

- **Edge Popularity Value (EPV)**: The cumulative number of times, across all clients of the application server, a causal occurrence has occurred between the vertices HBV and HAV.

If there already exists a link between HBV and HAV then the associated edge's EPV is incremented by one. This provides a scheme within which the most popular edges will maintain the highest values. However, this may not reflect the current popularity of the causal relations, therefore the EPVs purpose is to help in pruning edges from the graph. Periodically, the graph is searched and edges with an EPV below a given threshold result in their removal. After a round of reconfiguration all remaining edges have their EPV set to zero.

Periodic edge removal and the resetting of EPVs provides the contention management framework with a basic reconfiguration process to more appropriately reflect the current semantic and causal relationships between shared data items. The drawback of reconfiguration over employing a static graph is the performance cost relative to the number of vertices and edges in the graph. The decision on the reconfiguration period must be based on two factors: (i) the relative performance cost of reconfiguration; and (ii) the number of client requests within the period. If the number of requests is low but reconfiguration is too frequent then edges may be removed that are still popular but simply have not seen enough accesses in the period. As such, the reconfiguration period must dynamically change given the changes in load.

3.6. Client Injection Rate Modification

The description of the framework has so far been entirely a server-side modification. The backoff and contention management is managed by each application server and only requires for a client to roll back when requested. When considering scenarios, such as the high frequency trading case study, where client update frequencies are high, not involving the client in the contention framework will lead to a high level of rollback required. Given that a client can perform updates at a greater rate when compared to

communication latency and any required backoff period, the amount of updates a client will need to undo will be quite large. To reduce the overhead incurred from the rollback method of contention management, the client must also be part of the framework by adjust the injection rate of updates. The injection rate modification is used to either slow down or speed up the update frequency to match the ability for the application server to reduce the overall number of conflicts.

To vary injections rates of a per client basis a rollback threshold is employed. A client can determine the degree of rollback as the number of updates, originally applied to the local replica, that must be undone when instructed by the application server to rollback. Each client maintains a rollback threshold value which, if exceeded after determining the degree of rollback, triggers the modification of the injection rate. As the degree of rollback has exceeded the threshold the injection rate is halved. Should a rollback message be received and the degree of rollback does not breach the threshold then the injection rate value is increased by one. As this process continues for every rollback message received from the application server a steady injection rate will eventually be achieved at each client.

4. Simulation and Results

The previous chapter presented a description of the model for the contention management of distributed applications. To determine the effectiveness of the proposed model, this chapter presents a description of the discrete event simulation created. From this simulation, results are then presented describing the performance of the contention management scheme for a number of different metrics.

4.1. Simulation Architecture

The simulation framework was implemented using a discrete event simulation framework library called SimJava [Sim]. The processes of the model (clients, application servers, load balancer and database) are simulated as communicating entities. Communication between entities in the simulation is performed by scheduling an event. An event has an associated delay to simulate the communication latency that would be experienced in a typical distributed application operating over the Internet. After the delay has expired the event will be delivered to the intended recipient's event queue. The message queue of an entity is processed in first-in, first-out order. Self-events are used in each entity to schedule periodic events. A self-event is simply an event that is scheduled by an entity whose recipient is that same entity. This event will be delivered to the entities after the specified delay and will be processed as events are consumed from the entities event queue.

Before each experiment, all entities are created and set up with the required data and parameters. Each client entity is provided with the generated data accesses and is connected to the load balancer such that all messages are directed to this entity. The load balancer accepts messages from all clients on the same event queue and directs these to the application server entity responsible for that client. Each application server

entity is provided with the initial state of the graph and is connected to the load balancer and database entity. The database entity has the master copy of the data set and is connected to all application server entities.

4.1.1. Data Generation

Within the simulation, data relating to client updates and the contention graph structure are pre-generated for each experiment. The graph structure is generated based on the data items present at the database entity. Each data item is represented as a vertex in the graph while the initial edge structuring can be configured in one of two ways. A fully configured graph structure has a set of vertices where each vertex is connected to every other vertex. This option is useful in situations where the causality of user updates is not initially known. Periodic reconfiguration of the graph will quickly remove a large number of the redundant edges and more accurately reflect the current causality. The second option for graph creation is to provide two configuration parameters that allow for a structure to be produced. These two parameters are a maximum number of edges per vertex and a probability of adding an edge to the vertex. For each vertex in the graph an edge is added if the probability is not exceeded and the current vertex does not already have greater than the maximum edges. The result, depending on the parameters, is a varied graph structure used as the basis for the experiment.

Given the initial graph structure, client access patterns are generated. Each client has a different set of accesses generated following the structure of the graph. To model a break in the causality of a client, there is a configurable chance parameter that determines whether the next access to be generated follows the structure of the graph or a unrelated vertex is chosen as the next update for that client.

4.1.2. Load Balancer

The load balancer ensures that there is an even distribution of clients across deployed application server entities. A single load balancer is used in all experiments. As entity failure is not modeled or is the focus of the experiments, the load balancer does not need to redirect clients; once a client has been assigned to an application server, all requests will be sent to the same server.

4.1.3. Clients

A number of client entities can be deployed as required for the required experimental scenario. A client generates an update based on the current data item in its access pattern list. This access triggers an update to the logical clock value in the client's local replica of the state. Once a message is sent by a client entity the state change to the local replica is recorded in the client's execution log such that rollback can be performed if requested by the application server. Before a client can send the next access update to the application server, any messages received must be processed first. If required, the client will rollback using the execution log to revert the logical clock values of updated data items to their original values.

4.1.4. Application Server

Each application server is provided with the initial configuration of the graph for the experiment. The application server has the following jobs to manage:

- **Processing client messages**: The required checks are made for the requested update by the application server entity. Any communication that is required with the database is performed as synchronous communication. An event is scheduled with the required request to the database entity and the application server entity blocks until the response event is received from the database.

- **Managing the delta queue**: The delta queue is not explicitly modeled as a queue with elements but instead each element, that is the client being backed off, is modeled as a self-event within the application server entity. When a backoff period is generated for the client, a self-event is created and the delay is set as the generated period. During this period, the server ignores any messages sent from the client as session conflicts. Once the self-event delay has expired, the event will be delivered to the application server entity, who will process it and generated the appropriate backoff request for the client.

- **Graph Reconfiguration**: Reconfiguration of the graph is scheduled using a self-event with the delay as specified by the reconfiguration parameter of the experiment. After reconfiguration has occurred, the application server will schedule another self-event for the the next reconfiguration.

4.1.5. Database

In all experiments, a single database entity is deployed. The database entity manages the master copy of the data set. It accepts requests by the application server entities for logical clock values and processes any state changes also requested by an application server.

4.2. Results

In this section, results are presented in the context of the three case studies presented previously: e-commerce, high frequency trading and online gaming. These domains provide a context for the parameter values used to configure the simulation and illustrate the impact on the contention management for that given domain based on it's requirements.

4.2.1. Parameters and Configuration

A number of key parameters determine the performance of the contention manager:

- **Clients**: Client processes are the source of contention in a distributed application where optimistic replication is employed. A system where a single client process operates would exhibit no contention or conflicts. By adding a second client, the chance for a conflict to occur increases and so does the overall system contention. As a metric, the number of clients provides a level of contention for the application from which a further two metrics can be evaluated.

- **Conflicts**: A conflict occurs when a client process updates an out of date data item. This is detected by the application server process and triggers the contention management scheme in the form of the back off and roll back mechanism. There is no expectation that a contention manager can remove contention from a system; the aim is to manage contention as optimally as possible. This can be measured as a reduction in the number of conflicts in comparison to the absence of any contention management scheme. The number of conflicts presented in this results section are taken as a total across all application servers.

- **Throughput**: Throughput is measured as the number of successful updates applied at the database per second. As a successful contention manager aims to

reduce the number of conflicts, this in turn also allows a greater level of throughput. The backoff periods for clients allows for an access schedule to be developed, reducing the chance for clients to access out of date versions of the data item. In comparison, without a contention manager, clients processes would consistently be accessing out of date data items and throughput would be much lower. The throughput presented in this results section are taken as an average across all application servers providing an overall system throughput.

To be able to compare and evaluate the contention management scheme, a number of protocol configurations exist:

- **No Contention Management (No CM)**: The contention management scheme is not enabled in this setting. The application server responds to a conflicted client request immediately and does not perform any back off. The message informing the client of a conflict contains the state update and logical clock for the data item the client conflict with and requires the client to roll back. This message does not contain the preemptive state update as the semantic contention management is not employed.
- **Contention Management (CM)**: The contention management is enabled in this setting. The protocol operates where by the client is backed off in the event of a conflict.
- **Contention Management with Reconfiguration (CM + Recon)**: As above, but the dynamic reconfiguration is enabled for the graph.

The configuration without contention management is required to provide a base line for the experiments. It is to be expected that, with the introduction of the contention management scheme, that the number of conflicts are reduced and throughput is improved over a system without contention management. From this, the introduction of reconfiguration of the graph can be compared to that of just contention management to gauge the impact that reconfiguration has on both throughput and conflicts.

4.2.2. Case Study: E-commerce

Setup

In this set of experiments, the three protocol configurations are explored with relation to the the number of conflicts and overall throughput. Table 4.1 presents protocol parameters used for this case study:

Parameter	Value
Edges per data item	10
Prediction list length	5
Frequency increment for target data item	4
Frequency increment for neighbouring item	2
Frequency ceiling	200
Reconfiguration period	20 seconds
Edge removal threshold	1
Latency	50 - 150 milliseconds
Accesses per client	200
Updates per second	10

Table 4.1.: Experiment parameters

The parameters to configure the structure of the graph were chosen to reflect the domain. In these scenarios, the semantic graph is identifying relationships seen in the buying habits of customers across a range of products. An arc between two data items reflects a purchase made for these two products within one transaction. Therefore, the edges per data item has been set high at 10 to reflect a wide range of choice the end user has when it comes to the relationship between the products. In relation to this, the prediction list length is set at 5 to reflect that a large number of data items may be bought within a single transaction. By setting this higher, the client is provided with a more update to version of the data as held by the database. Frequency values were chosen based on the number of accesses and the latency as these are contributing factors to the back off periods. The reconfiguration values used are to support a larger number of reconfigurations. This is to reflect the requirement of changing buying habits that can occur within this domain. The edge removal value is set low to identify any changes where there a product may no longer be a popular purchase in relation to other items. Latency values have been chosen based on figures provided in [Ver]. These values are very hard to judge as they can vary greatly depending on a number of configurations such as world location and network hardware. The chosen range was come to based on

the statistics provided and supports expected latencies seen for global communication. Within the simulation, these values are measured as a random variable with normal distribution between 50 - 150 milliseconds. The number of accedes is set to provide a long enough simulation and allows for reconfiguration to occur a number of times.

Results

For this case study, results are presented for the three described protocol configurations. The first set of graphs show the conflict and throughput values for no contention management and contention management. The next set then compare the results of contention management with contention management including reconfiguration. A single graph size (500 data items) is presented here. As expected, the graph size does not have a significant impact on the conflicts or throughput. Each graph displays the average of 50 executions of the experiment; the same parameters are used but different seeds for the simulation.

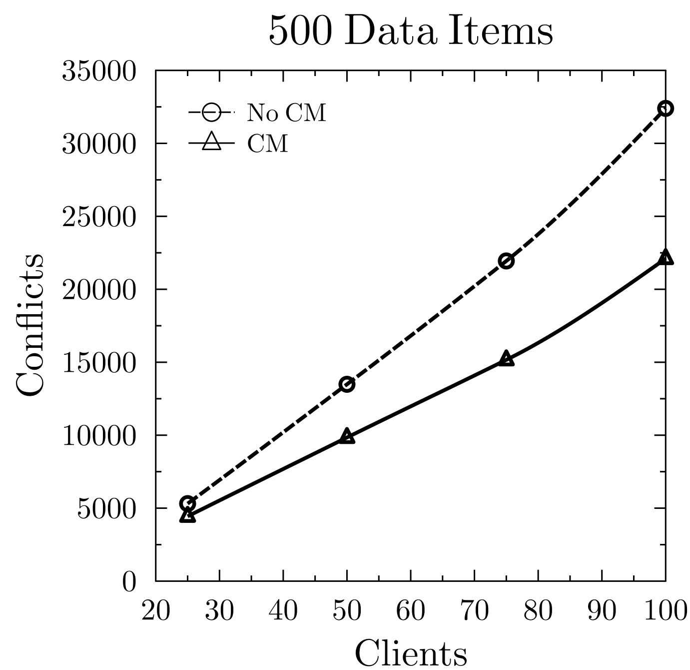

Figure 4.1.: Conflicts: with and without contention management

Clients	25	50	75	100
No CM - Mean	5311	13488	21941	32396
No CM - SD	11.28	12.35	12.01	14.7
CM - Mean	4440	9842	15163	22103
CM - SD	13.83	14.72	13.98	15.03

Table 4.2.: Conflicts: Statistics for 50 executions

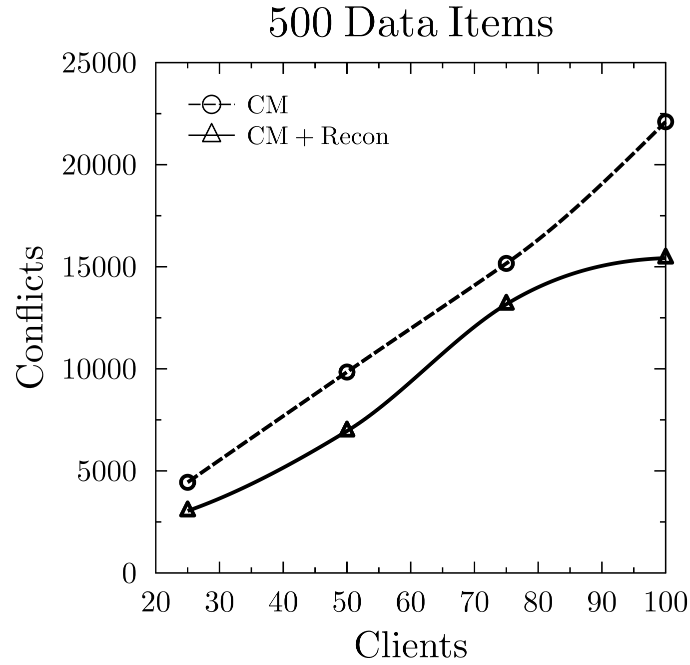

Figure 4.2.: Conflicts: contention management and reconfiguration

Clients	25	50	75	100
CM - Mean	4440	9842	15163	22103
CM - SD	13.83	14.72	13.98	15.03
CM + Recon - Mean	3039	6952	13162	15425
CM + Recon - SD	12.73	12.15	13.05	13.77

Table 4.3.: Conflicts: Statistics for 50 executions

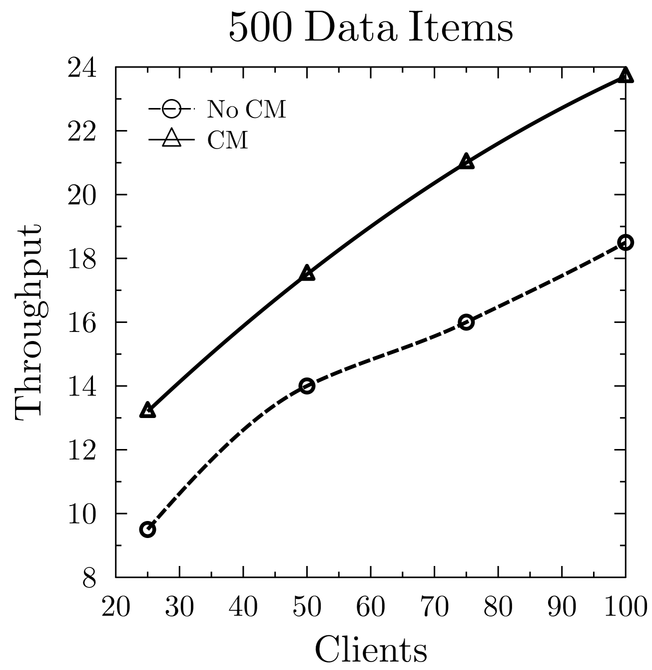

Figure 4.3.: Throughput: with and without contention management

Clients	**25**	**50**	**75**	**100**
No CM - Mean	9.5	14	16	18.5
No CM - SD	1.63	1.57	2.06	1.96
CM - Mean	13.2	17.5	21	23.7
CM - SD	1.52	1.7	1.23	2.25

Table 4.4.: Throughput: Statistics for 50 executions

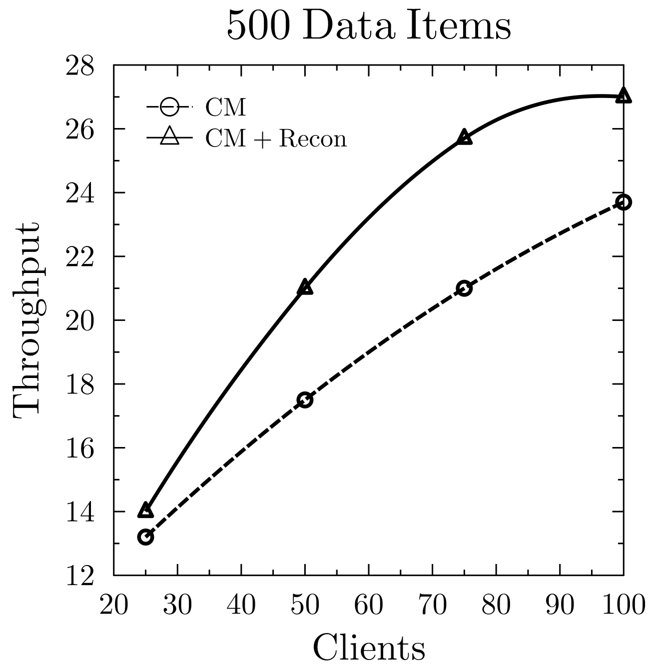

Figure 4.4.: Throughput: contention management and reconfiguration

Clients	25	50	75	100
CM - Mean	13.2	17.5	21	23.7
CM - SD	1.52	1.7	1.23	2.25
CM + Recon - Mean	14	21	25.7	27
CM + Recon - SD	1.73	1.15	1.07	1.78

Table 4.5.: Throughput: Statistics for 50 executions

Analysis and Discussion

These results show that the use of a contention manager has provided a reduction in conflicts and increase throughput. The difference between no contention management and contention management for conflicts becomes more apparent as the number of clients increase. This shows that the greater the contention (larger number of clients), the better the contention management scheme can scale successfully. These results are statistically significant ($P < 0.05$) and support the validity of the contention management approach in this scenario.

The addition of contention management, as shown in graph 4.3, shows that reconfiguration also has a favourable impact on the number of conflicts. The difference is

not as great when comparing the no contention management condition with contention management, but is still enough to warrant the use of reconfiguration as a means to improve performance. What is interesting to note is how, as reconfiguration approaches 100 clients, the number of conflicts start to level out. This indicates that reconfiguration supports system stabilization whereby the graph configuration has reached a point whereby the contention is not changing.

Throughput values in graph 4.4 show improvement when using contention management. The different is not quite as large as would perhaps be expected; this is due to the absolute values of the conflicts being large. This is a performance improvement and shows that the contention management does not have a negative impact on the throughput. It is interesting to see that the use of reconfiguration also provides a further improvement in throughput, as show in graph 4.5. The same sort of system stabilization is seen in these throughput results as was seen for the conflict values when reconfiguration is enabled. This is to be expected as the number of clients rises to a point whereby no more bandwidth is available to support more operations being serviced.

4.2.3. Case Study: High Frequency Trading

Setup

In this set of experiments, the no contention management and contention management with reconfiguration protocol configurations are explored with relation to the the number of conflicts and overall throughput. The reason for choosing reconfiguration to be included for contention management is based on the previous results shown for e-commerce. The results show that reconfiguration has a positive impact on the system and as such should be included. In this case study, the use of client injection modification is used. This case study is interested in identifying if injection rate modification can provide further benefits to the number of conflicts and throughput. This type of system experiences very large numbers of client requests whereby the resulting contention is high. If a client is allowed to submit update requests as quickly as possible, the level of contention will be very high, resulting in a large number of conflicts. A threshold value is used for injection rate modification. The results here are shown for two threshold values; 30 and 60. Table 4.6 presents protocol parameters used for this case study:

Parameter	Value
Edges per data item	3
Prediction list length	2
Frequency increment for target data item	8
Frequency increment for neighbouring item	4
Frequency ceiling	200
Reconfiguration period	40 seconds
Edge removal threshold	1
Latency	25 - 50 milliseconds
Accesses per client	200
Updates per second	100

Table 4.6.: Experiment parameters

The parameters to configure the structure of the graph were chosen to reflect the domain. In these scenarios, the semantic graph is identifying relationships seen in trades between a number of stocks. An arc between two data items reflects a trade made for these two different stocks within one transaction. The edges per data item has been set low at 3 to reflect a low number of common relations that exists between stocks due to the nature of the domain. In relation to this, the prediction list length is set at 2 to reflect that a lower number of trades being made within one transaction. Frequency values were chosen based on the number of accesses and the latency as these are contributing factors to the back off periods. A larger number of requests are expected and so these values are required to suit the expected contention. The reconfiguration values used reflect a reduced requirement for reconfiguration based on the assumption that the relationships between data items are well known for this type of system. Latency values have been chosen as per the requirements of these type of trading systems. Low latencies support a more up to date view of the data. However, there is no literature to support what typical values are required in these scenarios. These values have been chosen based on the assumptions and system requirements expected. Within the simulation, these values are measured as a random variable with normal distribution between 25 - 50 milliseconds. The number of accedes is set to provide a long enough simulation and allows for reconfiguration to occur a number of times. The updates per second have been set at an initial value of 100. This value will soon change as clients start to conflict and the injection rate modification starts to stabilise the client's injection rate.

Results

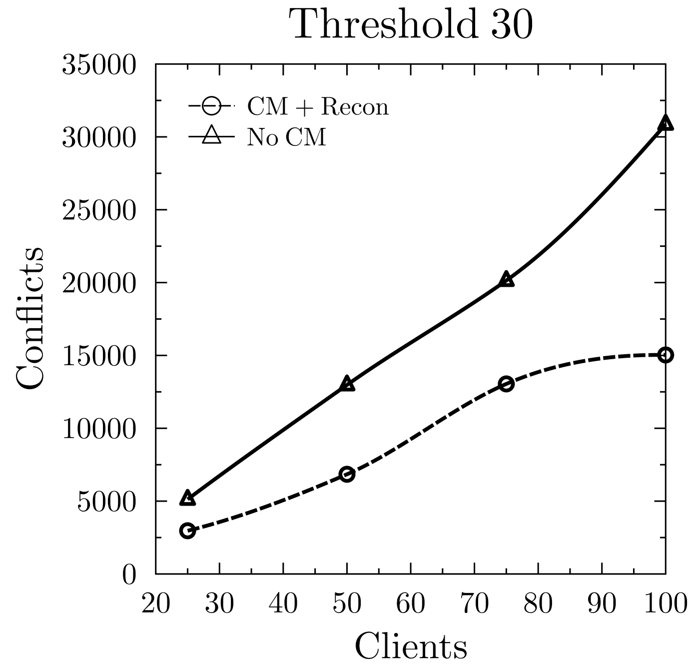

Figure 4.5.: Conflicts: with and without contention management (reconfiguration), threshold 30

Clients	25	50	75	100
No CM - Mean	5123	12975	20143	30864
No CM - SD	10.57	11.75	12.87	15.41
CM + Recon - Mean	2967	6843	13045	15032
CM + Recon - SD	12.84	15.76	12.09	14.61

Table 4.7.: Conflicts: Statistics for 50 executions, threshold 30

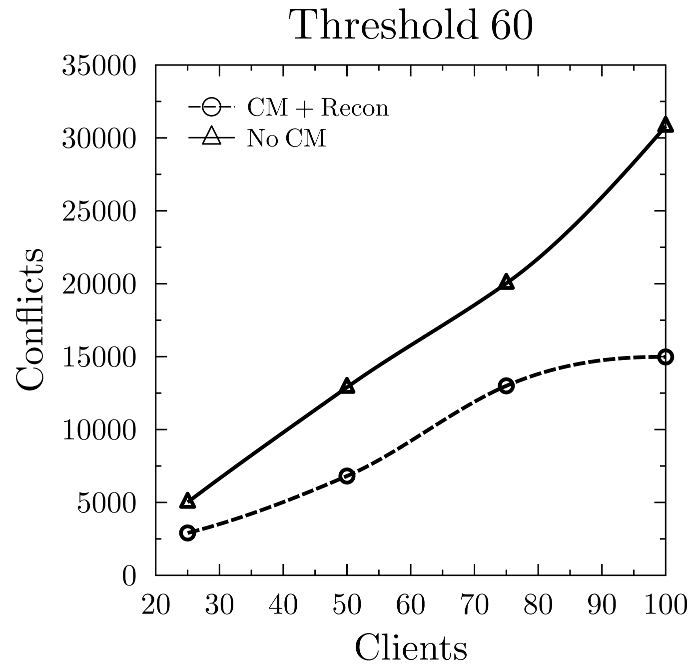

Figure 4.6.: Conflicts: with and without contention management (reconfiguration), threshold 60

Clients	25	50	75	100
No CM - Mean	5013	12895	20026	30799
No CM - SD	12.47	13.47	12.57	14.12
CM + Recon - Mean	2907	6812	12997	14978
CM + Recon - SD	13.47	11.57	14.5	12.77

Table 4.8.: Conflicts: Statistics for 50 executions, threshold 60

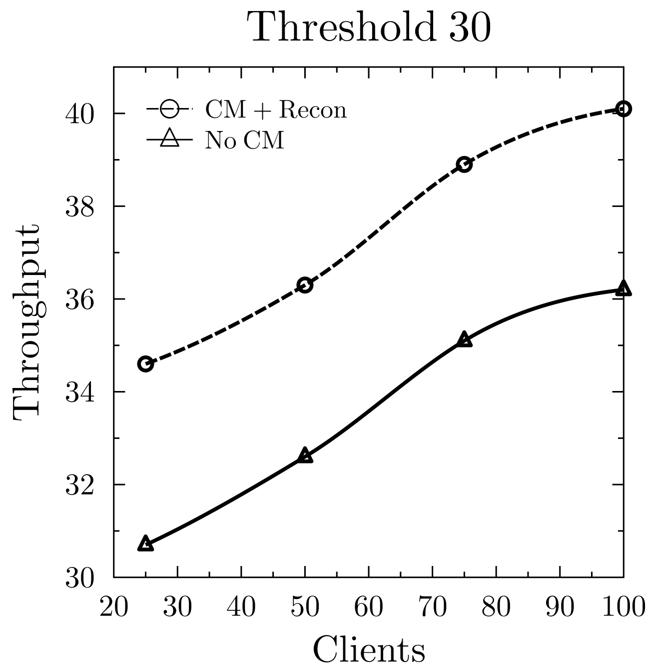

Figure 4.7.: Throughput: with and without contention management (reconfiguration), threshold 30

Clients	25	50	75	100
No CM - Mean	30.7	32.6	35.1	36.2
No CM - SD	1.75	1.21	1.86	1.76
CM + Recon - Mean	34.6	36.3	38.9	40.1
CM + Recon - SD	1.73	1.74	1.13	2.29

Table 4.9.: Throughput: Statistics for 50 executions, threshold 30

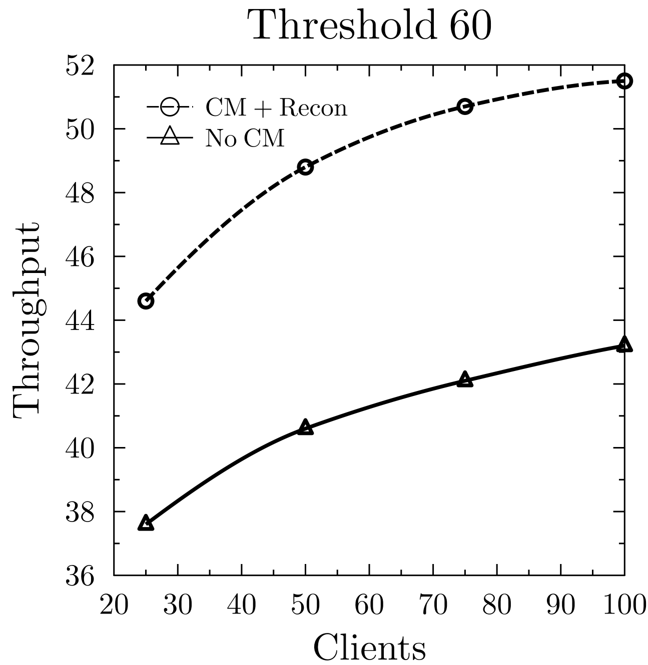

Figure 4.8.: Throughput: with and without contention management (reconfiguration), threshold 60

Clients	25	50	75	100
CM - Mean	37.6	40.6	42.1	43.2
CM - SD	1.71	1.47	1.72	1.25
CM + Recon - Mean	44.6	48.8	50.7	51.5
CM + Recon - SD	1.24	1.14	1.87	1.14

Table 4.10.: Throughput: Statistics for 50 executions, threshold 60

Analysis and Discussion

Overall, the use of injection rate modification has little impact of the number of conflicts but a large improvement for throughput. When comparing these results to the e-commerce scenario, the first point to consider is that the client updates per second parameter is higher for the high frequency case study. This value is set to 100 initially but will quickly be adjusted as this value will result in a large roll back for the client. In the e-commerce scenario, the value is set at a static 10 updates per second. This provides a base line from which the impact of injection rate modification can be assessed. The improvement in throughput comes from a greater injection rate at clients but what

can be seen is that, by managing this rate, there is no great impact to the number of conflicts. If a larger injection rate was used without any adjustment the number of conflicts would be much larger as clients are more likely to update out of date data items. These results show that the injection rate modification can be used to achieve a greater level of throughput without impacting the number of conflicts.

4.2.4. Case Study: Network Gaming

Setup

In this set of experiments, the no contention management and contention management with reconfiguration protocol configurations are explored with relation to the the number of conflicts and overall throughput. The reason for choosing reconfiguration to be included for contention management is based on the previous results shown for e-commerce and high frequency trading. Client injection modification is used in this case study as it has been shown to support good performance as seen in the high frequency scenario. This case study is interested in identifying whether latency has an impact of the throughput and number of conflicts. Latency values are of particular concern to real time simulation systems such as network gaming. As actions are performed locally, the server must determine whether these actions were valid with regards to the current simulation state. The issues of contention and local replication are very much relevant to the domain of online gaming and, as such, this presents an interesting area to investigate for the semantic contention management. Latency values have been explored within the literature [CHL06], [CC06]. From this existing work, two latency bands have been chosen to investigate: low latency (10 - 60 milliseconds) and high latency (100 - 300 milliseconds). Table 4.11 presents protocol parameters used for this case study:

The parameters to configure the structure of the graph were chosen to reflect the domain. In these scenarios, the semantic graph is identifying relationships between different objects in the game world and actions a player can perform. The data items represent these high level objects, while the arcs represent a relationship these objects have as per the player's interaction. The edges per data item has been set high at 10 to reflect a large number of possible actions a player has in relation to the object. In relation to this, the prediction list length is set at 8 to reflect the high level of

Parameter	Value
Edges per data item	10
Prediction list length	8
Frequency increment for target data item	8
Frequency increment for neighbouring item	4
Frequency ceiling	200
Reconfiguration period	40 seconds
Edge removal threshold	1
Accesses per client	200
Updates per second	100
Threshold	60

Table 4.11.: Experiment parameters

inconsistency a player can enter due to the real time requirements of this type of simulation. The number of accedes is set to provide a long enough simulation and allows for reconfiguration to occur a number of times. The updates per second have been set at an initial value of 100. This value will soon change as clients start to conflict and the injection rate modification starts to stabilise the client's injection rate. The threshold has been set to 60 due to the favourable results seen in the high frequency case study.

Results

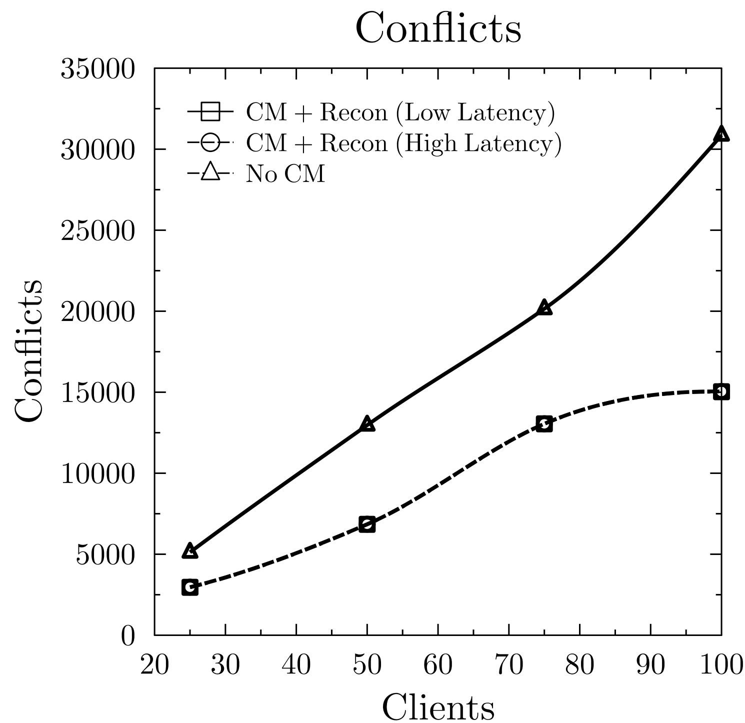

Figure 4.9.: Conflicts: with and with contention management (high and low latency)

Clients	25	50	75	100
No CM - Mean	5120	12965	20149	30863
No CM - SD	15.21	14.78	12.68	12.57
CM + Recon (Low Latency) - Mean	2966	6852	13050	15046
CM + Recon (Low Latency) - SD	12.56	13.54	13.78	14.87
CM + Recon (High Latency) - Mean	2957	6844	13047	15029
CM + Recon (High Latency) - SD	12.44	13.57	13.78	13.96

Table 4.12.: Conflicts: Statistics for 50 executions

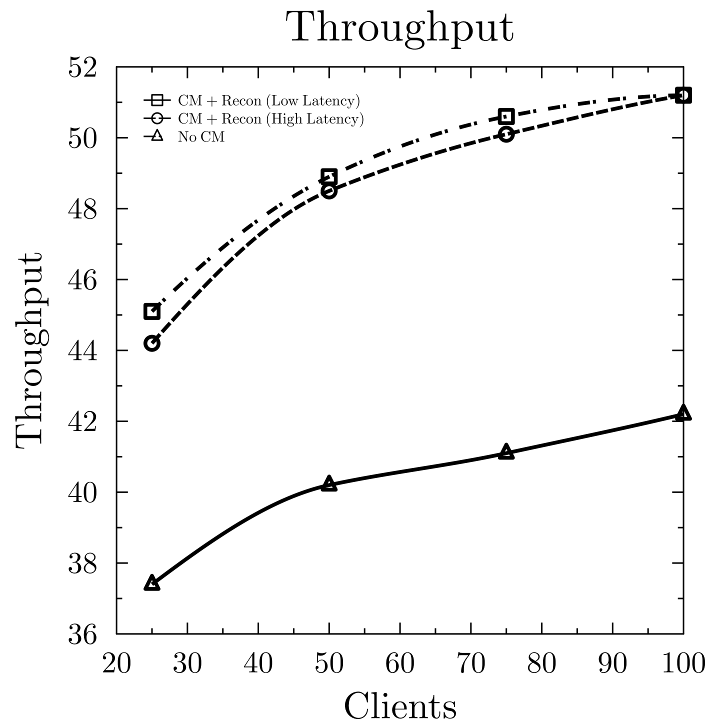

Figure 4.10.: Throughput: with and without contention management (high and low latency)

Clients	25	50	75	100
No CM - Mean	37.4	40.2	41.1	42.2
No CM - SD	1.43	1.42	1.12	1.85
CM + Recon (Low Latency) - Mean	44.2	48.5	50.1	51.2
CM + Recon (Low Latency) - SD	1.74	1.22	1.74	1.34
CM + Recon (High Latency) - Mean	45.1	48.9	50.6	51.2
CM + Recon (High Latency) - SD	1.85	1.45	1.69	1.67

Table 4.13.: Throughput: Statistics for 50 executions

Analysis and Discussion

From the graphs and figures, the latency differences have no impact on throughput or conflicts. The figures for the three protocol configurations are very similar to the high frequency case study. Initially it might be expected that the configuration with high latency may have a larger number of conflicts. Due to the message delay, a client will make more progress before finding out that roll back is required. When considering the operation of the protocol, however, the conflicts will not rise due to the latency. A conflict is detected and then a client is backed off. No more messages will be processed for that client. The impact of high latency on the client is actually seen in the degree of roll back that would be required. The longer a message takes to process, the more operations a client will perform locally before being informed about roll back.

The results here show that the contention management scheme can be employed within an environment where latencies values can vary greatly. As the performance remains consistent among the case studies, the solution is generic enough to support w wide range of different system requirement and configurations.

5. Conclusion

This chapter provides a summary of the thesis, discussing each chapter, and presents future work.

5.1. Thesis Summary

This thesis has explored the applicability of contention management techniques in the context of distributed applications, where optimistic shared data access is employed. These applications exhibit high levels of semantic links between data accesses that, while satisfied within the local replication of state, must also be maintained when integrated in the global consistency model. This requirements form a degree of semantic knowledge that is used to inform the contention manager how best to handle data access requests. The aim, like any optimistic shared data scheme, is to provide a higher level of throughput. This comes at a cost of managing any conflicts that arise from concurrent updates to data items.

The following chapter breakdown summarises the content of this thesis:

- Chapter 1 provided the introduction for the thesis. This included a discussion of distributed systems and their use of replication techniques in supporting availability and reliability. Separate to this, a discussion on the use of contention managers is also presented, highlighting their use within the domains of distributed multiple access and, more recently, transactional memory. The contribution of the thesis forms a link between these two topics of distributed applications and contention management by presenting an approach for managing contention within optimistic replication protocols for a range of distributed application types. A number of papers are listed where work from this thesis has been published.

- Chapter 2 provided the background and related work to the topics of distributed applications, replication and contention management. The focus of the background for distributed applications was scalable data which included an in depth look at the various consistency models. A move from strong to weak consistency models showed how sacrificing guarantees about the consistency can be made to support a greater degree of scalability. The notion of eventually consistent systems strikes a balance between the requirements of consistency and scalability required by the demands of modern systems that experience very large data access requirements. The topic of replication presented the pessimistic and optimistic approaches as described in the literature. The pessimistic approach favours a stronger consistency model while optimistic approaches support scalability at the cost of the potential for conflicts. Various techniques were described for how these conflicts are both detected and handled in optimistic replication systems. Contention management is presented in the context of its application within distributed multiple access and transactional memory systems. These two domains have seen significant improvements relating to throughput by deploying a contention manager to regulate access to the shared medium, in the context of distributed multiple access, and the shared data objects for transactional memory. These same kind of requirements are seen in the eventual consistent systems where a large number of processes compete for access to shared data. The related work focuses on optimistic replication protocols in the literature and the contention management policies used in transactional memory. The different policies used in transactional memory for contention management are shown to provide variable performance benefits depending on the application type. Chapter 2 also includes a description of three application types where a contention manager could be used to provide an access control mechanism for shared data. These application types all exhibit a level of optimistic replication but do not consider using a contention manager to achieve performance improvements in the literature. To conclude the background section, a gap analysis was presented. This analysis considered the systems described in the literature and summarised their contributions as specific characteristics that were deemed important to the context of this thesis. From this gap analysis, the thesis was able to determine the criteria required going forward into the design chapters.
- Chapter 3 presented a design for a n-tier distributed application including a

communication protocol to support a contention management framework deployed as a server-side extension. The client plays no part in the backoff contention management scheme, other than being required to rollback as requested, forming a transparent operation. The inclusion of further extensions to the design in the form of graph reconfiguration and client injection rate modification also allow for run-time adjustments to be made. The reconfiguration of the graph structure allows for a more accurate reflection of the current contention and access patterns to be captured at the server-side. Client injection rate modification allows for the system as a whole to react to different workloads requirements over the lifespan of the system.

- Chapter 4 described the implementation of the design outlined in chapter 3. This implementation took the form of a discrete event simulation following the same design as chapter 3. The simulation architecture discussed the roles of each process and the method of communication. The rest of this chapter provided the evaluation of the contention management framework in the context of the three application types described in chapter 2. The evaluation of the protocol, via simulation, demonstrates the effectiveness of the backoff contention management protocol across a range of different environments. When compared to a protocol where the backoff contention management is not used, the inclusion of contention management provides significant reductions in the number of irreconcilable conflicts while providing an improvement in the throughput of successfully committed updates. The use of graph reconfiguration also demonstrates its effectiveness as shown in the results. The use of client injection rate modification is shown to provide a self-regulating framework for run time adaption that provides favourable results. As expected, the drawback of a backoff based contention manager is the wasted execution performed by a client when backed off at the server side. A client will continue progressing, not knowing that some actions will already have to be rolled back. A modification to the protocol that sees the client play a role in the backoff scheme is one possible solution to this problem. When a client conflicts, the application server instructs the client to stop sending messages for the generated backoff period. The decision to make the backoff scheme server-side exclusive was made in the interest of allowing for the protocol to be applicable across a range of different environments without requiring the client to even be aware that

contention management was being employed. The results presented for each application type show that the use of a contention manager sees real world performance benefits in the context of access management for shared data.

5.2. Thesis Discussion

This work set out with the aim of applying the existing domain of contention management to the context of optimistic replication exhibited within distributed applications. Although both domains were well understood, the use of a contention management had not been explored as a possible means of managing the conflicting accesses that are a typical problem for this area. To this end, a design and simulation was produced from which to determine if a contention manager could provide a performance benefit. As presented in section 2.8, in the context of their implementation and performance in this thesis, the requirements of a contention manager are to:

- **Manage access to shared resources**: This is the main aim of a contention manager and is achieved by providing scheduled access such that when a client comes to access the shared resource, they are less likely to be in a conflicted state. In this thesis, the contention manager was deployed as a server side mechanism. This allows for the server to generate a view of client accesses to support not only the semantic graph but also the overall contention. From this, by applying a back off period in which client updates are not processed, subsequent accesses have a greater chance of being accepted. By applying back off periods across all clients, a natural schedule is developed.

- **Reduce the number of conflicts**: The main of the contention manager is to reduce the number of conflicts. Conflicts are a typical problem seen in optimistic replication schemes and are difficult to manage. The results presented in this thesis show that contention management is very much a viable solution to reducing the number of conflicts within an optimistically replicated system. The use of semantic knowledge of the domain supports this, as contentious items are identified by their frequency of use. The graph mechanism supports this well as it also identifies the semantic relations that exist between the data items. The benefit of using semantic knowledge of the domain was shown to be successful in the existing literature and is clear from the results of this thesis.

- **Increase access throughput**: With a contention manager, processes attempt to access the resource with the only constraint on the speed in which they can process and communicate. This results in the server being overloaded with messages and wasted execution of client processes for those that cannot successfully update. The contention manager supports a greater level of successful throughput as a result of the managed access and the reduction in conflicts. Had the throughput improvement been marginal, this could have been tolerated due to the nature of the reduction in conflicting accesses. However, the improvement in throughput supports the performance of contention manager as a solution for shared data access.

The problem of managing shared access in the context of optimistically replicated systems is a very large subject area. Often solutions are bespoke and suitable for a small domain. This thesis has attempted to present an abstract domain of distributed applications and a number of mechanisms that could find use across a broad range of domains. The graph mechanism is sufficiently generic to be able to address a range of different semantics that go beyond frequency. A number of drawbacks are present with the approach described here. As the contention management mechanism is supported as a server side enhancement, clients naturally perform executions that will never be successful. As a client is backed off by the server, a client continues to operate unware that a message will be arriving from the server informing the process to roll back. The overhead of roll back must also be taken by the client to ensure any actions made locally are reverted. The alternative would be bringing the client processes into the contention management scheme. The server would need to inform the client that they must cease communication for the back off period. The decision was taken to support a server side enhancement based on the case studies presented and the aim of making the contention management scheme transparent to the client processes. The high frequency trading and online gaming scenarios would not support the notion of the client ceasing communication. This would manifest as asking a trader to pause trading or a player to cease playing the game.

5.3. Future Work

This section presents a number of potential avenues for future research based on the contribution of this thesis. This is the first time contention management has been explored in its application for distributed systems and future work could form a number of interesting PhD projects. The following describes a number of areas for future work:

A Deeper Exploration of the Probabilistic Model

This thesis employs a basic model of client accesses to form the relations between data items. This takes the form of the graph where vertices represent the data items and edges indicate a semantic link. The use of reconfiguration allows for a straightforward mechanism for maintaining the relations in the graph as a reflection of the dynamic accesses made by clients. To allow for a more formal description and analysis of the access relations in this model, a Bayesian network could be used in the model. This representation of the frequency of access in the systems would allow for more accurate probabilities to be associated with client accesses when the server is required to generate a backoff time or provide an authoritative roll back message where a client is pre-emptively updated.

Consideration of the Server-side Implementation

The use of contention managers in transactional memory is concerned with the contention below the application layer, while this thesis has focused on the specific contention seen at the application layer. When considering the server-side implementation, future work could view the issue of contention all the way from core to application, marrying the use of techniques seen in transactional memory at the core level with the access preservation and contention management introduced in this thesis. There are a number of considerations for this work in relation to the different threading models present, the contention management policies deployed and the level of semantic links present in the application.

Existing and New Application Domains

This thesis has focused on the three domains of e-Commerce, High Frequency Trading and Massively Multiplayer Online Games. Each exhibits a level of contention and has been shown to benefit from a contention manager. Within these three areas there is

scope for yet more work and experiments to be performed, to determine the most effective parameters, given differing workloads. As the contention managers policies in transactional memory have shown, there is scope for dynamic deployment of different policies to suit a change in the workload of the system. As such, there may be ways to tailor the contention management so that additional performance improvements can be found in different operational environments. The three domains presented here also do not define the limits to where a contention manager can be deployed. A potential area of future work could focus on mobile environments, where replication is put to use to satisfy the requirements of transient network connectivity.

A Practical Implementation

The implementation of the contention manager in this work has been based in simulation. This has provided favourable results, providing scope to take this forward and explore how this could be implemented in a practical real world environment implemented in .NET or J2EE. This would introduce a number of additional issues to consider, such as managing failover of system processes, and supporting modern replication strategies as seen in Cassandra.

Bibliography

[ANB+95] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, 1995.

[BDSZ94] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 212–225, New York, NY, USA, 1994. ACM.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, New York, 1987.

[Bre00] Eric A Brewer. Towards Robust Distributed Systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 7–10, 2000.

[BZ91] Brain N Bershad and Matthew J Zekauskas. Midway : Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical report, Technical Report CMU-CS-91-170, Carnegie-Mellon University, 1991.

[CC06] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, November 2006.

[CHL06] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. How sensitive are online gamers to network quality? *Commun. ACM*, 49(11):34–38, November 2006.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory Access Buffering in Multiprocessors. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 14(2):434–442, 1986.

[EG89] C A Ellis and S J Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD iIternational Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[FGC+97] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-based Scalable Network Services. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 78–91. ACM, 1997.

[Fid88] Colin J Fidge. Timestamps in Message-passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66, 1988.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD iIternational Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management. In *Distributed Computing*, pages 303–323. Springer, 2005.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[Her91] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[KCZ92] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 210–218, New York, NY, USA, 2001. ACM.

[KS92] James J Kistler and M Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[LS88] Richard J Lipton and Jonathan S Sandberg. PRAM: A Scalable Shared Memory. Technical report, Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, 1988.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[PPR+83] Jr. Parker D.S., Gerald J Popek, G Rudisin, A Stoughton, B J Walker, E Walton, J M Chow, D Edwards, S Kiser, and C Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[Pri08] Dan Pritchett. BASE: An ACID Alternative. *Queue*, 6(3):48–55, May 2008.

[PSM03] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-Based Reconciliation for Collaborative and Mobile Environments. In Robert Meersman, Zahir Tari, and DouglasC. Schmidt, editors, *On the Move to Meaningful Internet Systems 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55. Springer Berlin Heidelberg, 2003.

[PST+97] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[SE98] Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[Sim] SimJava - University of Edinburgh. SimJava. Accessed at: http://www.icsa.inf.ed.ac.uk/research/groups/hase/simjava/.

[SKK+90] Mahadev Satyanarayanan, J J Kistler, P Kumar, M E Okasaki, E H Siegel, and D C Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SKSM03] Nah-Oak Song, Byung-Jae Kwak, Jabin Song, and M E Miller. Enhancement of IEEE 802.11 Distributed Coordination Function with Exponential Increase Exponential Decrease Backoff Algorithm. In *Proceedings of the 57th IEEE Semiannual Vehicular Technology Conference*, volume 4, pages 2775–2778, 2003.

[SS04] William N Scherer III and Michael L Scott. Contention Management in Dy-

namic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[SS05a] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.

[SS05b] William N Scherer III and Michael L Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[ST97] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

[TDP+94] D B Terry, A J Demers, K Petersen, M J Spreitzer, M M Theimer, and B B Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

[Tho79] Robert H Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TTP+95] D B Terry, M M Theimer, Karin Petersen, A J Demers, M J Spreitzer, and C H Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[Ver] Verizon Enterprise Statistics. IP Latency Statistics - Verizon Enterprise Statistics. Accessed at: http://www.verizonenterprise.com/about/network/latency/.

[Vog09] Werner Vogels. Eventually Consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[WCP+02] Haitao Wu, Shiduan Cheng, Yong Peng, Keping Long, and Jian Ma. IEEE 802.11 Distributed Coordination Function (DCF): Analysis and Enhancement.

In *Proceedings of the IEEE International Conference on Communications*, volume 1, pages 605–609, 2002.

A. Protocol Pseudo Code

This section describes the proposed system model as a set of pseudo-code algorithms; one client algorithm and a number of server algorithms dealing with message handling, graph maintenance and contention management.

A.1. Notation

Table A.1 describes the important notations used throughout this chapter and is presented as a reference.

A.2. Pseudo-Code

The **send** primitive is used to indicate a process sending a message to another process. For example, if *as* is an application server, the statement "**send** [UMR, *values*] **to** *as*" describes the sending a message of type UMR, with contents *values*, to process *as*. A message has a type that helps the process clarify how to process the message. A client can send an Update Message Request (UMR) to the application server and receive a

Notation	Description
$\mathrm{C_i}$	A client process, labeled $\mathrm{C_1}$, $\mathrm{C_2}$, $\ldots \mathrm{C_i}$.
$\mathrm{AS_j}$	An application server process, labeled $\mathrm{AS_1}$, $\mathrm{AS_2}$, $\ldots \mathrm{AS_j}$.
$\mathrm{DI_k}$	A data item residing at the database, labeled $\mathrm{DI_1}$, $\mathrm{DI_2}$, $\ldots \mathrm{DI_k}$.
$\mathrm{CDC_{C_i}}$	The client data item logical clock the client, $\mathrm{C_i}$, has operated against.
$\mathrm{CSC_{C_i}}$	The client session clock for the client's current session.
$\mathrm{CAC_{C_i}}$	The client action clock used to track the order of messages.
$\mathrm{SI_{C_i}}$	The application server's view of the session clock of $\mathrm{C_i}$.
$\mathrm{AC_{C_i}}$	The application server's view of the action clock of $\mathrm{C_i}$.
$\mathrm{LC_{DI_k}}$	The logical clock value of $\mathrm{DI_k}$received from the database.

Table A.1.: Notations.

Missed Message Request (MMR) or Enhanced Authoritative Rollback Message (EARM) from the application server (with the opposite being true for the application server). The application server can also send a ClockRequest message to the database requesting the logical clock value for a given data item. A ClockResponse will be received by the application server for a previous ClockRequest message.

The values that are sent as part of the message are accessed using subscript. For example, m_{CDC}, where m is a message, indicates accessing the client data clock of the received message. As communication is asynchronous, messages are received to queues at processes. A process determines that a message has been received if the membership of the queue is not equal to the empty set. The **queue** primitive indicates that the contents should be processed in a first-in, first-out ordering.

The **rollback** primitive is used when an application server requests a client process to rollback due to a conflict. When a client is instructed to rollback, the execution log is used to undo any changes made after the action clock the server has specified. If the client has received an EARM then the updates contained in the message must also be made to the local data set. The **apply** primitive indicates that a client must modify the state and clocks of the specified items. The **list of** primitive identifies an array of elements of the specified type. For example, the statement "**list of** Integer AC" indicates an array of Integer values called AC.

The application server manages a Graph type containing a set of Vertex and Edge types. All of these types have a number of utility methods used to simplify the pseudo-code implementation.

A.2.1. Client Protocol

When a client process wishes to send an update to the application server, it executes the function described in algorithm A.1. Before a client can send a new update, any messages received from the server must be first processed (line 1 and 2). The session identifier received in the message is compared with the client's own session clock (line 3). If the received clock is greater then the message is processed; otherwise, it is ignored as it is a duplicate. The client updates it's own session clock such that any new messages sent are part of the new session (line 4).

The client now determines the type of message received. The message will either be a missed message request (line 5) or a enhanced authoritative rollback message (line 8). If the type is a MMR then the client rolls back the state of it's local replica to the specified action clock using the execution history (line 6). The maintenance of the execution history is not presented in this pseudo-code to enhance readability. If the type is a EARM then the client again rolls back it's local replica and also applies the partial state update to the replica (lines 9 and 10).

If no messages have been received from the application server then the client is free to send the new update message. The client's increases it's action clock (line 15) and sends an update message request containing the data item accessed, the data item's logical clock, the client's session clock and action clock (line 16).

Algorithm A.1 Client Execution

```
function sendUpdate(Data Item di, Integer CDC)
    Application Server as          /* the application server to send to */
    Integer CSC                    /* client session clock */
    Integer CAC                    /* client action clock */
    Local Replica DS               /* client's local replica */
    queue of EARM messages         /* received messages */

 1: if messages ≠ ∅ then
 2:   for all m in messages do
 3:     if m_si > CSC then
 4:       CSC = m_si
 5:       if m_type = MMR then
 6:         rollback DS to m_AC
 7:       end if
 8:       if m_type = EARM then
 9:         rollback DS to m_AC
10:         apply m_updates to DS
11:       end if
12:     end if
13:   end for
14: else
15:   CAC = CAC + 1
16:   send [UMR, di, CDC, CSC, CAC] to as
17: end if
```

A.2.2. Server Protocol

The server protocol is split into five separate algorithms to provide clarity for the reader. An application server has three main responsibilities:

1. Processing client messages, including generating backoff periods for conflicting clients and graph maintenance. This is covered by algorithms A.2, A.3 and A.4.

2. Reconfiguring the graph structure. This is covered by algorithm A.5.

3. Updating client's after a backoff period has expired. This is covered by algorithm A.6.

Algorithm A.2 covers the processing of a message received by the application server from a client. To simplify this pseudo-code segment the issue of handling missed messages is not presented. A message is handled based on the status of a client (lines 1 and 13). If a client was previously in the progress state then the application server requests the logical clock, for the data item accessed by the client, from the database (lines 2 and 3). A logical clock conflict check is made on line 4 to determine if the client accessed the most up to date version of the data item. If the clocks are not equal then the client is considered conflicted. The session identifier the application server holds for the client is increased, the client's status is set to stalled and the client is backed off (lines 5, 6 and 7). If there is no conflict then the data item is updated at the database and the graph is updated (lines 9 and 10). If the client was previously in the stalled state then the session of the client and the value the server holds are compared (line 14). The operation of the algorithm from this point onwards is the same as if the client was in the progress state and previously described. The only exception that the client, on successfully updating the data item, is moved to the progress state (line 21).

Algorithm A.2 Application server processing a client message

function serverReceiveMessage(Message m, Client c)
  Database *database* /* *the database holding the master data set* */
  list of Status *status* /* *current status of each client* */
  list of Integer AC /* *server's view of CAC* */
  list of Integer SI /* *server's view of CSC* */

1: **if** $status[c] = progress$ **then**
2:   **send** [ClockRequest, m_{di}] **to** *database*
3:   **receive** [ClockResponse, *clock*] **from** *database*
4:   **if** $clock \neq m_{CDC}$ **then**
5:     $SI[c]$ += 1
6:     $status[c] = stalled$
7:     backoffClient(c, m)
8:   **else**
9:     **send** [ClockUpdate, m_{di}, $m_{CDC} + 1$] **to** *database*
10:     updateGraph(c, m)
11:   **end if**
12: **end if**
13: **if** $status[c] = stalled$ **then**
14:   **if** $m_{CSC} = SI[c]$ **then**
15:     **send** [ClockRequest, m_{di}] **to** *database*
16:     **receive** [ClockResponse, *clock*] **from** *database*
17:     **if** $clock \neq m_{CDC}$ **then**
18:       $SI[c]$ += 1
19:       backoffClient(c, m)
20:     **else**
21:       $status[c] = progress$
22:       **send** [ClockUpdate, m_{di}, $m_{CDC} + 1$] **to** *database*
23:       updateGraph(c, m)
24:     **end if**
25:   **end if**
26: **end if**

Algorithm A.3 is called as part of the previous message processing algorithm when a client is in conflict and requires backing off. The backoff period is generated given the frequencies of the data item that was the focus of the client's update along with the frequencies of the most volatile vertices up to three hops away in the graph. As such, the frequency of the focus data item is found (lines 2 and 3). The most volatile neighbouring vertex is now found in the graph (line 5). If the current vertex has no outgoing edges then this will return null at which the loop is terminated (line 7). If a vertex was found then its frequency is added to the running total and this becomes the focus vertex and the loop iterates. Finally, this client is backed off by being added to the delta queue for the generated period (line 13).

Algorithm A.4 is called when the client has successfully updated a data item. The graph

Algorithm A.3 Generating a backoff period for a client

```
function backoffClient(Client c, Message m)
    Graph g                                   /* application server's graph */
    Delta Queue dg                            /* application server's delta queue*/
    Integer hops = 3                          /* number of hops in the graph */

 1: Integer counter = 0
 2: Vertex current = g.getVertex(m_di)
 3: Integer period = current.getfrequency()
 4: while counter < hops do
 5:   current = g.getMostVolatileNeighbour(current)
 6:   if current = null then
 7:     counter = hops
 8:   else
 9:     period += current.getfrequency()
10:     counter += 1
11:   end if
12: end while
13: add c to dg for period
```

needs to be updated by incrementing the frequencies of the data items and recording the edge use for reconfiguration. The data item that was the focus of the client's action has it's frequency incremented (line 3). All vertices that are neighbours to the focus data item also have their frequencies updated (lines 4 - 7). Given the last update the client performed, the graph is checked to determine if an edge exists between this and the focus of the client's current update (line 9). If no edge exists (the getEdge() method returned null) then an edge is added between these two vertices (line 11). If there is already an edge then it's EPV is incremented by one (line 13). Finally, the current data item the client has updated becomes the HBV (line 15).

Algorithm A.5 is called periodically by the application server to perform reconfiguration of the graph. The reconfiguration involves removing any edges from the graph that have not seen a number of accesses greater than the specified threshold. For all the edges existing in the graph, the EPV is compared with the threshold (line 3). If the edge's EPV is lower than the threshold then this edge is removed from the graph (line 4). If the edge has seen a number of accesses greater than the threshold then the edge's EPV is reset to zero (line 6).

Algorithm A.6 is called whenever a client's back off period expires after being in the application server's delta queue. Now that the period has expired, the client's needs to be updated and removed from the delta queue (line 1). The partial state update is also

Algorithm A.4 Updating the graph

```
function updateGraph(Client c, Message m)
    Graph g                                   /* application server's graph */
    Integer increment                         /* value to increase frequency by */
    list of Vertex HBV                        /* application server's happens before values */

 1: Vertex current = g.getVertex(m_di)
 2: Integer old = current.getfrequency()
 3: current.setfrequency(old + increment)
 4: list of Vertex neighbours = g.getNeighbours(current)
 5: for all Vertex n ∈ neighbours do
 6:     old = n.getfrequency()
 7:     n.setfrequency(old + increment)
 8: end for
 9: Edge e = g.getEdge(HBV[c], current)
10: if e = null then
11:     g.addEdge(HBV[c], current)
12: else
13:     e.incrementEPV()
14: end if
15: HBV[c] = current
```

Algorithm A.5 Reconfiguring the graph

```
function reconfiguration()
    Graph g                                   /* application server's graph */
    Integer threshold                         /* edge weight threshold */

 1: list of Edge edges = g.getAllEdges()
 2: for all Edge e ∈ edges do
 3:     if e.getEPV() < threshold then
 4:         g.removeEdge(e)
 5:     else
 6:         e.resetEPV()
 7:     end if
 8: end for
```

required; the required data items are found using the graph (line 2) and are requested from the database (lines 3 and 4). These are then sent to the client (line 5).

A.2.3. Injection Rate Modification

Algorithm A.7 presents modifications made to the client side protocol as described in algorithm A.1. The degree of rollback is calculated (line 9) and used to determine if the threshold has been exceeded (line 12). If the threshold has been exceeded then the injection rate is halved (line 13) otherwise the injection rate is increased by one (line 15).

Algorithm A.6 Delta queue expiration

```
function deltaQueueExpiry(Client c, Data Item di)
    Graph g                              /* application server's graph */
    Delta Queue dq                       /* application server's delta queue */
    list of Integer AC                   /* server's view of CAC */

 1: remove c from dq
 2: list of Data Items updates = g.getStateUpdateItems()
 3: send [ClockRequest, updates] to database
 4: receive [ClockResponse, clocks] from database
 5: send [EARM, AC[c], clocks] to c
```

Algorithm A.7 Client side injection rate modification

```
function sendUpdate(Data Item di, Integer CDC)
    Application Server as                /* the application server to send to */
    Integer CSC                          /* client session clock */
    Integer CAC                          /* client action clock */
    Local Replica DS                     /* client's local replica */
    Integer threshold                    /* client's injection rate threshold */
    Integer rate                         /* client's injection rate */
    queue of EARM messages               /* received messages */

 1: if messages ≠ ∅ then
 2:   for all m in messages do
 3:     if m_si > CSC then
 4:       CSC = m_si
 5:       if m_type = MMR then
 6:         rollback DS to m_AC
 7:       end if
 8:       if m_type = EARM then
 9:         Integer rollbackDegree = calculateRollbackLength()
10:         rollback DS to m_AC
11:         apply m_updates to DS
12:         if rollbackDegree > threshold then
13:           rate = rate / 2
14:         else
15:           rate = rate + 1
16:         end if
17:       end if
18:     end if
19:   end for
20: else
21:   CAC = CAC + 1
22:   send [UMR, di, CDC, CSC, CAC] to as
23: end if
```