DELAY TOLERANT LAZY RELEASE CONSISTENCY FOR DISTRIBUTED
SHARED MEMORY IN OPPORTUNISTIC NETWORKS

by

CHANCE RAY EARY

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2011

Copyright © by CHANCE RAY EARY 2011

All Rights Reserved

To my father Ray, for his unerring support.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor, Mohan Kumar, for his contributions, feedback and patient counseling while writing this thesis, as well as being my advocate within the department and assisting me with financial support over the course of my graduate studies. I would also like to thank David Levine and Jeff Lei for agreeing to serve on my thesis committee.

Bahram Khalili was instrumental in my admission to graduate school, and I sincerely appreciate his believing in my abilities and potential.

And I would like to thank my father for always being there for me through the difficult times and acting as a voice of reason to reach me through my (very occasional) stubbornness.

July 15, 2011

ABSTRACT

DELAY TOLERANT LAZY RELEASE CONSISTENCY FOR DISTRIBUTED SHARED MEMORY IN OPPORTUNISTIC NETWORKS

CHANCE RAY EARY, M.S.

The University of Texas at Arlington, 2011

Supervising Professor: Mohan Kumar

Opportunistic networking (ON), a concept which allows a mobile wireless device to dynamically interact with other wireless devices in its immediate vicinity, is a field with great potential to improve the utility of itinerant computational platforms. While ONs increase a device's ability to interact with its peers, the fleeting and intermittent connections between devices make many traditional computer collaboration paradigms, such as distributed shared memory (DSM), virtually untenable. DSM systems, developed for traditional networks, rely on relatively stable, consistent connections among participating nodes. In an ON, connectivity among nodes is distributed in time and space making it useful for delay tolerant applications only. When devices are disconnected their ability to collaborate is interrupted and their computing power underutilized until such time as they encounter their computing partners again. Because nodes within an ON are assumed to have no advance knowledge that they will encounter a specific device in the future, periods of disconnection between nodes could be lengthy or potentially infinite.

This thesis proposes delay tolerant lazy release consistency (DTLRC) for implementing distributed shared memory in opportunistic networking environments. DTLRC permits mobile devices to remain independently productive while separated and provides a mechanism for nodes to regain coherence of shared memory if and when they meet again. DTLRC allows nodes to make use of results produced on shared memory while disconnected. Simulations using experimentally determined data traces demonstrate that DTLRC is a viable concept for enhancing cooperation among mobile wireless devices in an opportunistic networking environment.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
ABSTRACT	v
LIST OF FIGURES	ix
LIST OF TABLES	xi
Chapter	Page
1. INTRODUCTION	1
2. BACKGROUND	3
2.1 Opportunistic Networks	3
2.2 Distributed Shared Memory	5
2.2.1 Existing Consistency Schemes	6
2.2.2 Limitation of Existing Consistency Schemes in Opportunistic Networks	14
3. DELAY TOLERANT LAZY RELEASE CONSISTENCY	16
3.1 Goals	16
3.2 Architecture	17
3.2.1 Write Collision	18
3.2.2 Collision Resolution	20
3.2.3 Write History	21
3.3 Operation	25
3.3.1 Creating Diffs	25
3.3.2 Retaining Diffs	26
3.3.3 Identifying Write Collisions	26
3.3.4 Examining Write History	28

3.3.5	Collision Resolution	31
3.3.6	Applying Writes to Shared Memory	38
3.3.7	Flushing Diffs	39
3.3.8	Adapting Collision Resolution Protocols	40
3.4	Social Cache	40
4.	SIMULATION AND ANALYSIS	44
4.1	Data Trace	44
4.2	Experiments	46
4.2.1	Setup	46
4.2.2	Uncontested Write Propagation	48
4.2.3	Shared Memory: One-to-One	49
4.2.4	Shared Memory: Groups	51
4.2.5	Shared Memory: All-to-All	56
4.3	Social Cache	57
5.	DISCUSSION AND CONCLUSION	60
5.1	Example Application	60
5.2	DTLRC Benefits	61
5.3	DTLRC Drawbacks	62
5.4	Social Cache Benefits	62
5.5	Social Cache Drawbacks	63
5.6	Future Work	63
5.7	Conclusion	64
	REFERENCES	65
	BIOGRAPHICAL STATEMENT	67

LIST OF FIGURES

Figure	Page
2.1 A multihop opportunistic network between smart phones	4
2.2 A distributed dataset within an opportunistic network	6
2.3 Sequentially consistent series of writes and reads	7
2.4 Sequentially inconsistent series of writes and reads	8
2.5 Causally consistent, but sequentially inconsistent	9
2.6 Not causally consistent	9
2.7 Eager release consistency	11
2.8 Lazy release consistency	13
3.1 Sample Memory Configuration	19
3.2 Cluster Example	21
3.3 A historical collision occuring between nodes C and E at time t_3 . . .	22
3.4 History Synchronization	24
3.5 Write Collision	29
3.6 Historical overlap at time t_3 ; both node B and node C have previously seen each other's current <i>write</i>	32
3.7 Collision Resolution Flowchart	39
3.8 Social Cache	41
3.9 Social Cache Data Requests	42
4.1 Simulated sharing scenarios	48
4.2 <i>Write</i> Propagation in Exp 1	50
4.3 <i>Write</i> Propagation in Exp 2	51
4.4 <i>Write</i> Propagation in Exp 3	53

4.5	<i>Write</i> Propagation in Exp 6	54
4.6	Theoretical and Practical Minimum State for Groups of Five Nodes	56

LIST OF TABLES

Table		Page
4.1	Data Trace Details	45
4.2	Exp 1 Nodal Separation Duration	46
4.3	Exp 2 Nodal Separation Duration	46
4.4	Exp 3 Nodal Separation Duration, in seconds	46
4.5	Exp 6 Nodal Separation Duration, in seconds	47
4.6	Average Time for <i>Write</i> to Propagate to Every Node	49
4.7	Average State and Collisions for One-to-One Sharing Over Ten Trials	52
4.8	Time Lived for Each Experiment Over Ten Trials	52
4.9	Average Collisions for Each Group Size Over Ten Trials	52
4.10	Time Lived for Each Group Size in Exp 1 Over Ten Trials	52
4.11	Time Lived for Each Group Size in Exp 2 Over Ten Trials	52
4.12	Time Lived for Each Group Size in Exp 3 Over Ten Trials	55
4.13	Time Lived for Each Group Size in Exp 6 Over Ten Trials	55
4.14	Average State for Each Group Size Over Ten Trials	55
4.15	Average State and Collisions for All-to-All Sharing Over Ten Trials	56
4.16	Time Lived for All-to-All Sharing Over Ten Trials	57
4.17	Nodes Utilized in Social Cache Simulations	57
4.18	Social Cache Interval Between Requesting and Receiving Data Over Ten Trials	58
4.19	Social Cache Time Lived Over Ten Trials	58

CHAPTER 1

INTRODUCTION

Opportunistic networking (ON) is a set of concepts wherein wireless devices, primarily mobile, can interact with other wireless devices in their immediate vicinity. As devices move about their environment and enter one another's transmission range, they can create a temporary, point-to-point connection between themselves. This connection presents an opportunity for processes to exchange data and collaborate with no dependence on pre-existing networking infrastructure or foreknowledge of where a device will be and when.

Many technologies developed to facilitate collaboration between processes over network connections, such as distributed shared memory (DSM), were intended for use with consistent and stable links between nodes. Because wireless devices in an opportunistic network are mobile, connections among them are erratic and susceptible to being dropped without warning. Devices have no ability to determine when, or if, they will again meet to resume collaboration.

Distributed shared memory has been a research topic within computer science for several decades. Central to all models proposed thus far is that each process participating in the system will have consistent access to shared memory, be it located on the same physical machine or available across a network connection. As DSM systems evolved, different methods to ensure the consistency of shared data were proposed. These schemes began to incorporate mechanisms to increase a system's tolerance to network delay, but still assumed participating nodes and their data would

ultimately be accessible when necessary. Within an opportunistic network, neither of the aforementioned assumptions is applicable.

To facilitate the collaboration of mobile wireless nodes in the presence of unpredictable and intermittent network connections, we propose delay tolerant lazy release consistency (DTLRC). DTLRC has two goals:

- Permit two or more processes to work with one another in the absence of a stable network connection, and
- Ensure that a node can continue working on shared data even if it is separated from its peers for extended periods of time.

DTLRC allows nodes in an opportunistic network to collaborate when connected to one another, but remain productive when disconnected from one another.

To assess the viability of DTLRC, several simulations were conducted. These simulations made use of experimentally collected data traces to present a realistic opportunistic networking environment. Through various scenarios, we demonstrate that maintaining consistency of shared memory among nodes using only brief, erratic connections is achievable.

In this thesis, Chapter 2 gives an introduction to opportunistic networking and a discussion of existing DSM systems. Chapter 3 provides an overview of the theory and algorithms used in DTLRC. Chapter 4 discusses simulations and analysis, and Chapter 5 includes a discussion and concluding remarks.

CHAPTER 2

BACKGROUND

2.1 Opportunistic Networks

An opportunistic network (ON) is created when mobile devices, either carried by people or attached to vehicles, come within point-to-point transmission range of other wireless devices. Once these devices are within transmission range of one another, they have the opportunity to create a dynamic, peer-to-peer network among themselves. If multiple wireless devices are present in a given area, nodes can work together to relay messages between two devices that are outside of either's direct transmission range, but share common neighbors. Opportunistic networks show great potential in increasing the utility of wireless devices by allowing them to work with other wireless platforms in their immediate vicinity. The capability of any single node increases when that node can collaborate with other wireless devices to accomplish more collectively than any single device could accomplish on its own. No pre-existing networking infrastructure is required, and no fore planning is necessary [1] [2] [3] [4].

In some cases, these temporary networks can exist for days at a time with minimal interruption, for example, smart phones of family members are likely to be close to one another over a weekend. In other cases, two devices may communicate for mere moments and never see one another again, for instance wireless devices attached to automobiles traveling in different directions that have temporarily stopped at a traffic light. The uniting factor of the presented scenarios is that the wireless devices will have little to no indication of how long a connection to another device will last, and when or if the connection will ever resume.

Figure 2.1: A multihop opportunistic network between smart phones

In Figure 2.1, five smart phones have established a multihop opportunistic network. Any of the five devices can now send messages to any other device, either directly or via the use of intermediate nodes. For example, device *A* can send messages to device *E*, using nodes *C* and *D* to forward data.

When attempting to apply computing techniques intended for traditional wired networks over opportunistic networks, a number of key challenges are presented. Amongst the most difficult to address is the above mentioned problem of devices not knowing for sure how long other nodes within the ON will be available to participate in the network. Nodes may frequently join and leave the network with no warning and may not be seen again for lengthy periods of time, if they're seen again at all [2] [3]. Combine this with the inherent difficulties of wireless communication, for example intermittent delay and data loss due to the wireless medium [5], and a challenging network environment is created with a high per-message cost.

One method for executing processes to collaborate with one another over networks is distributed shared memory (DSM). If wireless devices within an ON are to make the most of their opportunistic contacts with other nodes, a method of employ-

ing DSM over an ON should be developed. We discuss distributed shared memory below, and explain some of the difficulties existing DSM methodologies might face if used on an opportunistic network.

2.2 Distributed Shared Memory

Distributed shared memory is a concept that entails multiple computational entities, either on the same physical system or connected via a network, to jointly operate on a set of data [6]. In most network-based DSM systems, each node will have a local copy of the entire dataset to eliminate the need to retrieve a value from a different node across the network each time it wishes to perform a memory operation. A node will make changes to its local copy of the dataset then notify other nodes of the new modifications to that memory, while updating the local copy of memory to reflect writes made by other nodes. DSM systems assume that there is no common clock available to all nodes, and due to network latency, write notifications may arrive at different nodes in different orders. Because of this, nodes must agree on a scheme to collectively determine how to apply writes to their local copy of shared memory. These schemes ensure that the local copy of shared memory will be consistent with the copy at other nodes.

In Figure 2.2, smart phones *A*, *B* and *E* have shared data across an opportunistic network. They can now collaboratively operate on that data and accomplish more collectively than any single device could independently.

DSM systems are designed for use over networks with relatively high reliability links between high availability nodes [7]. While DSM systems will accommodate a certain degree of network delay, loss of connection with a node due to a network failure will result in that node waiting to rejoin the network before it can continue working on its local copy of shared memory. The separated nodes will remain fully

Figure 2.2: A distributed dataset within an opportunistic network

functional while disconnected from one another, but their processing power cannot be utilized until the resumption of connectivity. Such disconnections are expected to be commonplace in opportunistic networks.

In order for distributed shared memory to work correctly, all processes participating in the system must have a consistent view of shared data. Consistency schemes are methods that allow processes to decide which value in the shared data is appropriate to use, and which value should be overwritten to reflect more relevant writes. Simply using the “most recent” write to determine which values should be used by all processes is not possible because nodes do not share a common clock [8]. We discuss existing consistency schemes below, and examine why they would or would not be appropriate for usage across an opportunistic network.

2.2.1 Existing Consistency Schemes

Consistency in distributed shared memory is a thoroughly researched topic within computer science. Originally intended for usage on multiprocessor computers,

the concepts have evolved to accommodate multiple processors on different computers which are networked together. We examine several well known consistency schemes below, and discuss their potential application in opportunistic networks.

2.2.1.1 Sequential Consistency

Amongst the earliest consistency schemes was sequential consistency proposed by Lamport [9]. Sequential consistency allowed a multiprocessor computer to interleave operations from processors in any order, with the operations of any one processor appearing in the order specified by its program. All processes would see the same interleaving of operations, with each individual process seeing the writes from all other processes while only seeing its own reads.

Figure 2.3 illustrates a sequentially consistent series of reads and writes, and figure 2.4 shows a series of memory operations that are not sequentially consistent (figures are from [6]).

In order to guarantee correct execution with this scheme, all memory requests have to be serviced from a single first-in first-out queue. A processor can often be ready to perform its next operation on memory but be forced to wait until subsequent requests, either from itself or from other processes, has been serviced from the queue.

Figure 2.3: Sequentially consistent series of writes and reads

If deployed over a network, sequential consistency would be very susceptible to performance degradation due to network delay, as all processes must see the writes

Figure 2.4: Sequentially inconsistent series of writes and reads

of every other process before having their own memory operation serviced by the central first-in first-out queue. Having a centralized FIFO queue to service every process gives a single point of failure, should the coordinating node crash or lose its network connection to other nodes participating in the system.

Within an opportunistic network, where each message has a high cost, sequential consistency would be expensive to maintain, as each node within the network must see the writes of every other node before proceeding with its own operations. Though the interleaved operations form a partial-order, each node is still forced to wait for the result of a preceding operation even if that operation has no relevance to that particular node. Furthermore, frequent disconnections expected within an ON could result in every node waiting for the result of an operation from a node that has lost its connection with the network.

2.2.1.2 Causal Consistency

Causal consistency, proposed by Hutto and Ahamad [10], was intended to allow greater concurrency between processes deployed over a network. DSM architects had observed that the traditional criterion of “reads return the most recent writes” was not well-defined within distributed systems. The absence of a global clock and latency inherent in networks meant there was no way to insure any single write was, in fact, the most recent. Causal consistency was meant as a step towards recognizing acceptable executions that fell outside the bounds of the standard definition of correctness.

In causal consistency, if event ‘b’ was caused or otherwise influenced by event ‘a’, the scheme requires that all processes first see ‘a’, then see ‘b’. This scheme differentiates between events that are concurrent (occur simultaneously but are independent of one another), and events that are causally related. Events that are concurrent may be seen in different orders on different machines, but events that are causally related must be seen in the same order by all participating processes.

Figure 2.5 shows a series of memory operations that adhere to causal consistency, but not sequential consistency. Figure 2.6 illustrates read and write operations that are not causally consistent. Figures 2.5 and 2.6 have been derived from [6].

P1:	W(x)k			
P2:	R(x)i	W(x)j		
P3:	R(x)i		R(x)k	R(x)j
P4:	R(x)i		R(x)j	R(x)k

Figure 2.5: Causally consistent, but sequentially inconsistent

P1:	W(x)i			
P2:	R(x)i	W(x)j		
P3:			R(x)j	R(x)i
P4:			R(x)i	R(x)j

Figure 2.6: Not causally consistent

Causal consistency is better suited to employment over a delay-tolerant network than sequential consistency, as it acknowledges that not all writes are of consequence to all processes. With causal consistency, the importance of writes being transmitted over the network as quickly as possible begins to take on slightly less importance. The scheme increases a process’s ability to continue to produce useful results when

confronted by network latency, but still requires that all writes be proactively sent to all processes even if a process has no need to see that data.

Within the context of an ON, where nodes can become completely disconnected from one another, causal consistency’s ability to identify operations that are causally related is ultimately not useful. If two nodes are not connected there is no way that writes from one node can be seen by the other node, or have any effect on that node’s operations. Furthermore, causal consistency’s requirement of proactively sending all writes to every node regardless of whether or not that node needs that data would result in many superfluous messages being transmitted over the ON, with a high cost for each message.

2.2.1.3 Eager Release Consistency

Eager release consistency (ERC, originally known simply as “release consistency”), first proposed by Gharachorloo, et al [11], is a scheme that guarantees the consistency of data only at specific points during a program’s operation. The programmer is responsible for writing a “properly labeled program” that correctly designates when shared data will reflect the most recent available changes and to enter and exit sections at appropriate times to ensure consistency.

In contrast to the above consistency schemes, ERC associates segments of memory, in the form of pages, with locks that allow a process to have exclusive access to that page. Processes exchange those locks amongst themselves rather than having each process operate on the shared data as a whole. A critical section is a portion of memory to which multiple processes get exclusive access, sequentially. Eager release consistency works on the principle that if a process has entered a critical section, and thus has the exclusive ability to operate on data protected by the critical section, no other process can rely on that data being consistent until the first process exits the

critical section. Because a process cannot operate on shared data until it enters a critical section, there is no need for a process to see the most recent changes to the data because it is currently not allowed to operate on that data anyway.

ERC only guarantees shared data will be consistent when a process enters the critical section, and only issues invalidation requests when that process exits the critical section. Acquiring and releasing locks to the critical section now becomes a “synchronization point” within the program, with other processes in the system receiving their invalidation notifications when the process holding the lock releases it. Figure 2.7 illustrates the exchange of consistency information with eager release consistency [12].

Figure 2.7: Eager release consistency

Eager release consistency’s ability to let the developer specify exactly when consistency data will be traversing a network is very useful in the context of opportunistic networks as connectivity between nodes operating in an ON is erratic and subject to change without notification. The developer now has synchronization operations organized into discrete, controllable events, allowing the scheme to first recognize changes in connectivity within the network and take the appropriate synchronization action to maintain coherency between the caches of the remaining nodes.

This scheme does suffer from the drawback of unnecessary messaging overhead, however. When a process exits the critical section, ERC dictates that all nodes sharing memory be notified of what data was changed. Because access to the critical section is still exclusive amongst the nodes, there is no need to preemptively invalidate a node's data until it enters the critical section; the data may be further modified by other processes before this node can begin its memory operations, or the node simply may not need to enter the critical section for some time. This results in messages being sent across the network that have no effect on a process's operation. Within an ON, where the cost of each message is assumed to be very high, a scheme that could further decrease the frequency that consistency information is distributed amongst nodes would be desirable.

2.2.1.4 Lazy Release Consistency

Lazy release consistency (LRC), as developed by Keleher [12], is an extension of eager release consistency with a number of enhancements to make the scheme better suited to operating in an environment with a high per-message cost. Keleher divides consistency protocols into two broad groups: eager protocols and lazy protocols. Eager protocols move data before a process actually needs it in an attempt to minimize latency. Lazy protocols delay moving consistency information until absolutely required, decreasing the amount of communication between processes on the network.

One way LRC decreases the amount of communication between processes is by sharing synchronization data only between the process which is currently operating in the critical section and the next process to acquire the lock to the critical section (this also permits the same process to reacquire the lock multiple times in succession without sending any data across the network). LRC then combines invalidation noti-

fications and the synchronization transfer into a single message because they occur at the same time. Intuitively, LRC allows the cache at other processes to become inconsistent until later synchronization accesses. In this way, LRC substantially reduces the consistency information needing to be sent over the network. Figure 2.8 shows the exchange of consistency information with lazy release consistency [12].

Figure 2.8: Lazy release consistency

Steps are also taken to avoid false sharing. False sharing occurs when the system is tracking accesses at a granularity larger than the actual size of the shared data item (i.e., the page contains several data items that are unrelated to one another). Because a process must gain sole access to this page before the page can be modified, and the page itself encompasses more data than the specific item a process is attempting to modify, multiple processes could contest ownership of the page when they're attempting to modify disjoint sets of data. LRC avoids this issue by permitting greater granularity within a page and allowing multiple concurrent writers. Each writer obtains locks to the specific data item within the page it wishes to write to and modifies only that portion of the page, removing the requirement that a process must gain sole access to the entire page.

When initiating the system, a process allocates a chunk of local memory to hold the data that is virtually shared. The memory is located at the same virtual address

on each machine. When the process is granted access to a portion of the page, it creates a twin, or copy, of the page and temporarily stores it in system space. When the modification of the page is complete, a run-length encoding of the differences between the two versions of the page, called a diff, is created. These diffs are then used to update other process' copies of that page. With exception of the initialization, all updates to shared memory are performed via diffs.

The greatest strength of lazy release consistency is that it reduces the amount of synchronization data traversing a network to a bare minimum. Within the context of an opportunistic network, where messages may have to make several hops across battery powered mobile devices before reaching their ultimate destination, LRC is well-suited to employment when end-to-end connectivity between communicating nodes can be facilitated by the available devices.

2.2.2 Limitations of Existing Consistency Schemes in Opportunistic Networks

As of this writing, one paper has addressed memory consistency in a mobile computing environment [13]. This paper focuses primarily on mobile ad hoc networks rather than opportunistic networks, and employed a version of lazy release consistency as the mechanism for maintaining coherence in shared memory amongst nodes.

Lazy release consistency might be well-suited for use within opportunistic networks under certain conditions. However, neither LRC nor any of the above consistency schemes can accommodate distributed computing when end-to-end connectivity between two nodes is completely lost. Nodes in an opportunistic network are expected to leave and join the network frequently over the course of normal operation. The available consistency schemes would leave one or more disconnected nodes idling in the interim, unable to produce useful results on shared data. As two specific nodes

may never have another opportunistic contact, their available computing power could be underutilized for lengthy periods while waiting for their partner nodes to reappear.

CHAPTER 3

DELAY TOLERANT LAZY RELEASE CONSISTENCY

This chapter discusses the goals, architecture and algorithms used with delay tolerant lazy release consistency for maintaining consistency of distributed shared memory in opportunistic networking environments.

3.1 Goals

The consistency schemes discussed in the previous chapter are a selected collection of a wide variety of methods to allow two or more processes, located on different machines, to collaborate. Their shared commonality is that the processes must have a consistent, stable network connection to one another. Within an opportunistic network, where connections are often fleeting, these schemes will be relegated to operating only when an end-to-end connection is available.

Due to this restriction on traditional consistency schemes, we propose delay tolerant lazy release consistency (DTLRC). DTLRC has two primary goals:

- Permit two or more processes to work with one another in the absence of a consistent network connection, and
- Ensure that a node can continue working on shared data even when it is disconnected from its peers for long periods.

The objective of DTLRC is to enable nodes in an opportunistic networking environment to collaborate when possible, but remain productive when separated.

Utilizing DTLRC, this thesis proposes Social Cache. Social Cache extends DTLRC by:

- Enabling a group of nodes which regularly have opportunistic contacts to distribute memory load amongst themselves, and
- Allow greater heterogeneity of participating nodes by loosening the memory configuration requirements of DTLRC.

The goal of Social Cache is to enhance the flexibility and processing potential of DTLRC when used with closely associated nodes.

3.2 Architecture

When two or more nodes agree to collaborate via DTLRC, they first set aside an area of memory for use in the system. The memory is uniformly divided into segments, with each segment assigned an index. The arrangement of segments and indices is identical for all participating machines, a scheme derived from [12]. The entire data set necessary for operation is then copied to each machine and stored in the predetermined area of memory. Memory requirements grows linearly with the amount of content to be shared in the system.

Five metadata values are associated with each index of shared memory:

- Cluster ID: if contiguous segments of shared memory are related to one another, they are assigned to a cluster. The cluster ID is an integer and is located at the same indices with the same ID on all nodes.
- Priority: an integer, initially set to zero. Should an application determine that a *write* is of high importance, an integer greater than zero can be assigned to the index. The maximum priority is set according to the requirements of the application.
- *Write* history: a list with the following values:
 - Origin: the node ID that created this *write*.
 - Creation: the internal time at the origin node when this *write* was created.

- Received: the time at the local node when this value was received from a peer. Will be the same as creation time if value written locally.

(In this thesis, a ‘write’ is the action of creating or modifying a value in memory; a ‘*write*’ is the encapsulation of this value for transport across the network.)

Figure 3.1 shows a possible memory configuration amongst three nodes. Here, the indices and cluster assignments are identical across all three nodes, and the write history is shown for index ‘30405’. The present *write* for all nodes at index ‘30405’ has originated from node *B*, with an elevated priority of ‘1’. Some values are held outside the table and located by their pointers ‘*’.

Because memory is identically arranged on all participating nodes, information about operations to memory can be communicated between nodes via *diffs*. ‘Diffs’, similar to those used in [12], are objects containing the metadata for indices in shared memory. The *diffs* can then be used to determine the most appropriate way to apply *writes* to shared memory when two nodes meet after being separated.

As each participating node in the system has a full copy of the entire dataset, it is free to leave and join the network at any time. While peers share a connection with one another, a traditional consistency scheme, such as lazy release consistency, can be used to accommodate collaboration. Once a node has determined it no longer has a connection to its peers, metadata for the most recent *writes* are committed to the node’s *write* history, and operation of DTLRC commences.

3.2.1 Write Collision

While a node is operating on its own, it is free to read and write to its copy of shared memory whenever required by the process. When a node makes a modification to shared memory, it stores the metadata for its *write* in its own *write* history. Any

Figure 3.1: Sample Memory Configuration

nodes with which this process was collaborating prior to being disconnected can also read and write freely to their copies of shared memory.

Because these nodes are unable to collaborate in the absence of a network connection, two or more nodes may write to the same index in their local copy of shared

memory. These conflicting modifications to memory are called ‘*write collisions*’. If nodes carrying a copy of shared memory do not encounter one another again, these *write collisions* are irrelevant and can be ignored; the system will continue working on its copy of memory, with the only additional overhead being the updates to the shared memory metadata. If nodes carrying shared memory do meet one another later, they will negotiate which *writes* to retain in a procedure called collision resolution, described in the next section.

3.2.2 Collision Resolution

DTLRC assumes a global clock (such as that provided by a global positioning system or cellular phone network) is not available to nodes. Because *writes* cannot directly affect one another while nodes have no end-to-end connection, simply determining which *write* was the ‘most recent’ may ultimately be meaningless, even if a global clock were available.

If a process has produced a *write* that has not resulted in a collision, that *write* is automatically selected for retention by the system. When a *write collision* does occur, three methods are used for selecting the most appropriate *write* to retain:

- Priority-based: if two *writes* to an index have different priorities, the *write* with the highest priority is selected for retention.
- Cluster-based: because segments assigned to a cluster are directly related to one another, *writes* to a cluster from two different nodes should not be mixed. The resolution protocol will select *writes* to this cluster from one node for retention.
- Volume-based: should the preceding criteria be equal or otherwise not applicable, the node that has produced the highest volume of *writes* will have its *write* retained in the event of a collision.

Figure 3.2 illustrates a cluster. Here, node *A* will show operations at indices 0 and 2, and node *B* will show operations at indices 1 and 2. If you were to take node *A*'s operations at 0 and 2, and mix them with node *B*'s operation at 1, the operations will produce an illogical result.

Node A			Node B			Result	
Index	Value		Index	Value		Index	Value
0	X = 4	←	0	X = 2		0	X = 4
1	Y = 3		1	Y = 1	←	1	Y = 1
2	Z = X*Y = 12	←	2	Z = X*Y = 2	←	2	Z = X*Y = 12

Figure 3.2: Cluster Example

By applying the collision resolution protocols, nodes can agree on a consistent view of memory. While only two nodes will synchronize at a time, those nodes can then propagate their agreed upon view of memory to other nodes participating in the system. As opportunistic contacts continue to occur, processes continue synchronizing the contents of their local version of shared memory, while assimilating new *writes* into the system.

3.2.3 Write History

Because two nodes can independently write to an index, and then independently share their *writes* with other nodes, it is possible to create a situation where a process has a current value overwritten by a previous value, carried by a different node. This is called a ‘historical collision’. To prevent nodes from having values repeatedly overwritten, a *write* history is kept for each index.

Figure 3.3 demonstrates a historical collision. At time t_0 , node A and node D have written a '1' and a '2' to the same index, respectively. At time t_1 , node A shares its write with node B , and node D shares its write with node C . At time t_2 , node B meets node C and a collision occurs; node C 's '2' is overwritten by the '1' from node B . At the same time, node D meets node E and shares its *write* for that index. At time t_3 , node E meets node C . Even though node C has previously encountered the *write* carried by node E , C accepts E 's value, overwriting the '1' that was selected for retention by the system. In this fashion, *writes* selected to be dropped by the system can circulate endlessly amongst participating nodes if left unaddressed.

Figure 3.3: A historical collision occurring between nodes C and E at time t_3

When an index is updated, the ID of the originating node is stored, as well as the local time at the origin node where the *write* was created and the time when the *write* was received at the local node. As nodes synchronize their local copies of memory, they check an incoming *write* against the *write* history for the affected index. If the same values for origin ID and creation time are found in the *write* history, the node knows it has seen this value previously. The collision resolution protocol would then select the node's local value to be applied to the shared memory of the remote process.

Synchronizing nodes will create a *write* history for both the value that was retained, as well as the value that has been overwritten. The *write* history of the *write* selected for retention is sorted to the first position in the list. One *write* history object for every node participating in the system is sufficient to ensure values that have previously been selected to overwrite are eventually removed from the local cache of all nodes. It is important to note that the timestamp maintained within the *write* history is only used as an indication of when the *write* was created when comparing later *writes* originating from the same node to the same index. If a later *write* to an index is seen from the same node, its existing entry in the *write* history is updated. Timestamps from two different nodes are only compared as static values associated with the *write*, and are used because they will be the same at any node currently holding the *write*; their value is not assumed to be an accurate indicator that one event preceded another.

Figure 3.4 depicts *write* history synchronization. The first panel shows time t_2 , after node B has written a '1' to index '51989'. When node A and node B meet at time t_3 , the system selects node A 's *write* for retention. Node B updates its *write* history to reflect the new value received from node A , and node A makes an entry in its *write* history for the value overwritten at node B . By recording the replaced

value from node *B*, node *A* will know to ignore this *write* should it be seen again from another node. Note that node *B* accepted node *A*'s *write*, even though it lists an earlier creation time; this is valid behavior in most cases (see section 3.3.4 for more details). Both nodes now have the *write* selected for retention at the first position in their list.

Figure 3.4: History Synchronization

The received time field is used to determine what length of time a node should notify its peers that the value in this index needs to be shared. Based on the rate of contacts between nodes in the system per period of time, DTLRC can select a

‘notification duration’ (ND). The ND is set as the estimated amount of time it would take a single *write* from a single node to propagate to a percentage of other nodes participating in the system if there were no *write* collisions. (On selecting an appropriate percentage of nodes to receive the write, please see Section 4.2.2). Upon an opportunistic contact, the process examines memory to determine which *writes* it holds have been received or created within the notification duration, and selects those *writes* for further consideration by its peers.

3.3 Operation

The operation of DTLRC focuses on identifying which values in memory need to be propagated through the system and resolving *write* collisions when they occur. The core functionality is discussed below.

3.3.1 Creating Diffs

When two collaborating nodes have another opportunistic contact, they both call *AnalyzeMemory* (Algorithm 1, page 25). *AnalyzeMemory* moves through the metadata for the local copy of shared memory, and examines the received time for each index. If the received time is within the designated notification duration, a *diff* is created for this index and added to a list. The resulting list of *diffs*, or diff list, is sent to the node initiated the opportunistic contact. The algorithms in this thesis identify the node which initiated the contact as ‘node 1’. The remote process is designated ‘node 2’. Here, ‘history.current’ is the history of the *write* that is currently being held in memory.

Algorithm 1

(* **AnalyzeMemory**: Check memory for modified segments. *)

Input: Pointer to shared memory, notification duration (ND), and the current time.

```

1.  LinkedList list  $\leftarrow$  new LinkedList
2.  for i  $\leftarrow$  0 to memory.length
3.      do
4.          if currentTime - memory[i].history.current.receivedTime  $\leq$  ND
5.              then diff  $\leftarrow$  new Diff(i, memory[i].cluster, memory[i].priority,
6.                  memory[i].history)
7.                  list.add(diff)
8.
(* Convert linked list to array. *)
9.  Diff array  $\leftarrow$  new Diff[list.length]
10. for j  $\leftarrow$  0 to list.length
11.     do
12.         array[j]  $\leftarrow$  list
13.         list  $\leftarrow$  list.next
14. return

```

3.3.2 Retaining Diffs

To keep track of which *writes* have been selected for retention, the initiating node creates a linked list of *diffs* called *retained* (Algorithm 2, page 27, line 1). To add *diffs* to the *retained* diff list, the process calls *Write*, and passes in the selected *diff*, the node from which this *diff* originated (either ‘1’ for the local node, ‘node 1’, or ‘2’ for the remote node, ‘node 2’) and the list of retained *diffs*. Once collisions are resolved, this list of retained *diffs* is shared between to the two contacting nodes, and updated values are exchanged.

When a *diff* is added to *retained*, the colliding *diffs* are removed from the *diff* lists of both nodes. This prevents *diffs* from accidentally being reexamined and reduces memory usage.

3.3.3 Identifying Write Collisions

Once the initiating node has created its own *diff* list and has received the *diff* list from the contacted node, it first checks the *diff* lists for *write* collisions by calling

CollisionResolution (Algorithm 2, page 27). Because the *diff* lists are created by sequentially examining each index in shared memory in ascending order, the virtual indices contained in the *diff* list will be sorted in ascending order.

Algorithm 2

(* **CollisionResolution**: Examines the diff arrays received during contact *)
 (* between nodes and resolves collisions. *)
Input: Array of diffs from Node 1 and Node 2.

1. LinkedList *retained* $\leftarrow$ new LinkedList
2. int *collision* $\leftarrow$ -1
3. **for** $i \leftarrow 0$ **to** *diff1*.length
4. **do**
5. **if** *diff1*[i] = null
6. **then** continue
7. (* *FindVirtualIndex* and *CheckCluster* will return the position of the colliding *)
8. (* write in the diff list of Node 2, if it exists. *Resolve*: Algorithm 4, page 31 *)
9. *collision* $\leftarrow$ FindVirtualIndex (*diff1*[i].virtualIndex, *diff2*)
10. **if** *collision* ≥ 0
11. **then** Resolve(i , *collision*, *diff1*, *diff2*, *retained*)
12. (* If the cluster was identified and resolved above, other elements of that cluster *)
13. (* will already have been removed from both *diff* lists. *)
14. **else if** *diff*[i].cluster > 0
15. **then** *collision* $\leftarrow$ CheckCluster(*diff*[i].cluster, *diff2*)
16. **if** *collision* ≥ 0
17. **then** Resolve(i , *collision*, *diff1*, *diffs2*, *retained*)
18. **else** Write(*diff1*[i], 1, *retained*)
19. **for** $j \leftarrow 0$ **to** *diff2*.length
20. **do**
21. **if** *diff2*[j] $\neq$ null
22. **then** Write(*diff2*[j], 2, *retained*)
23. **return** *retained*

As the resolution algorithm progresses, *diffs* succeeding the current loop position may be removed prior to the counter reaching that element within the *diff* array. When a new cycle of the loop in *CollisionResolution* begins, the process first verifies that the element has not been set to null earlier (page 27, line 5). Once the

element is verified to be present, ‘node 1’ examines the virtual index contained in this *diff*, by calling *FindVirtualIndex* (line 8).

FindVirtualIndex is a binary search that attempts to locate the same virtual index for the current *diff* at ‘node 1’ to a *diff* in the *diff* list from ‘node 2’. If the same virtual index is identified in both *diff* lists, a *write* collision has occurred, and the system calls *Resolve* (page 27, line 10). If the same virtual index is not contained in both lists, it is still possible that ‘node 1’ has written to a cluster and ‘node 2’ has written to the same cluster at a different index. If the *diff* from ‘node 1’ has a cluster value greater than zero, the system calls *CheckCluster* (page 27, line 13). *CheckCluster* will check the diff list from ‘node 2’ to ensure no *diffs* for that same cluster are present. If *CheckCluster* returns a value greater than zero, the system passes that value to *Resolve*. Should neither *FindVirtualIndex* or *CheckCluster* return a value greater than zero, an uncontested write has occurred, and the *write* is selected for retention by the system.

Figure 3.5 illustrates a *write* collision. Here, a *diff* lists from node ‘A’ and node ‘B’ are being examined during an opportunistic contact. Both *diff* lists contain an entry for index ‘20972’, and that *write* collision will need to be resolved. The other *diffs* shown in the figure are uncontested *writes*.

3.3.4 Examining Write History

To prevent a current *write* selected for retention by the system from being overwritten by a *write* previously selected for removal, the *write* history for this index is examined with *CheckHistory* (Algorithm 3, page 29). First, the process checks the most recent entry in the history of the *diffs* from ‘node 1’ and ‘node 2’; this entry will be the current *write* either node has in this index (page 29, line 3). If the most recent histories match one another, both nodes have the same *write* but

A	Index	Cluster	Priority	History
	1477	-1	0	*
	20971	9	1	*
	20972	9	1	*
	...	...	...	...

B	Index	Cluster	Priority	History
	17538	-1	0	*
	20972	9	0	*
	59220	-1	3	*
	...	...	...	...

Figure 3.5: Write Collision

have received it within the notification duration. The *diff* is removed from both *diff* lists, and *CheckHistory* returns ‘true’ to indicate a historical collision existed and was resolved.

Algorithm 3

(* **CheckHistory**: Examines the *write* history of this index at both nodes. *)

Input: Position of conflicting nodes in array lists from node 1 and 2, and the linked list of retained diffs.

Output: Boolean indicating a historical collision.

1. $i \leftarrow$ Position in diff array of node 1
2. $j \leftarrow$ Position in diff array of node 2
3. **if** $diff1[i].history.current = diff2[j].history.current$
4. **then** $diff1[i] \leftarrow null$
5. $diff2[j] \leftarrow null$
6. **return**
7. bool $found1 \leftarrow false$
8. bool $found2 \leftarrow false$
9. **for** $k \leftarrow 0$ **to** $diff2[j].history.length$


```

10.    do
11.        if  $diff2[j].history[k] = diff1[i].history.current$ 
12.            then  $found1 \leftarrow true$ 
13.                break
14.    for  $k \leftarrow 0$  to  $diff1[i].history.length$ 
15.        do
16.            if  $diff1[i].history[k] = diff2[j].history.current$ 
17.                then  $found2 \leftarrow true$ 
18.                    break
19.    if  $found1 = true$  and  $found2 = true$ 
20.        then
21.            if  $diff1[i].history.current.creationTime \geq$ 
22.                 $diff2[j].history.current.creationTime$ 
23.                then Write( $diff1[i]$ , 1, retained)
24.                else Write( $diff2[j]$ , 2, retained)
25.             $diff1[i] \leftarrow null$ 
26.             $diff2[j] \leftarrow null$ 
27.            return true
28.    if  $found1 = true$ 
29.        then Write( $diff2[j]$ , 1, retained)
30.             $diff1[i] \leftarrow null$ 
31.             $diff2[j] \leftarrow null$ 
32.            return true
33.    if  $found2 = true$ 
34.        then Write( $diff1[i]$ , 2, retained)
35.             $diff1[i] \leftarrow null$ 
36.             $diff2[j] \leftarrow null$ 
37.            return true
38.    return false

```

If the two *writes* are not the same, the process attempts to find the most recent *write* to this index from one node in the *write* history of the other node (page 30, line 11). If the most recent *write* from node *A* is located in the *write* history of node *B*, node *B* has encountered this *write* before. The *write* history of node *A* is also checked for the most recent *write* from node *B*.

Three outcomes are possible:

1. Both nodes have seen the other's current *write* previously (page 30, line 19).

This scenario is illustrated in Figure 3.6, where node *B* meets node *C* at time t_3 ,

and each node has the other's current *write* in its *write* history. In this case, the creation time for either *write* is examined. The *write* with the greatest creation time is selected for retention, and *CheckHistory* will return 'true'. It is important to note that creation time is used here simply as a value that will always be associated with this *write* for the duration of its presence in the system. It is used in this scenario because every node with this specific historical collision will resolve the collision the same way. It is not assumed that the "latest creation time" is an accurate indication one write occurred before another.

2. One node has seen the other's current *write*, but not vice versa (page 30, line 28). If node *A* has seen node *B*'s current *write*, node *B* will retain node *A*'s current *write*, and *CheckHistory* will return 'true'.
3. Neither node has seen the other's current *write*. *CheckHistory* will return 'false', and the *write* collision will be resolved via another method.

With these procedures, *writes* selected for removal from the system will eventually be dropped by all participating nodes.

3.3.5 Collision Resolution

If neither node has seen the other's current *write*, a collision resolution protocol is used to select which *write* should be retained by the system. The idea of collision resolution is to allow the nodes to agree on the state of shared memory in a way that is meaningful to the data contained therein. Collisions are processed and resolved by the *Resolve* algorithm (Algorithm 4, page 31).

Algorithm 4

(* **Resolve**: Selects which of the conflicted **writes** will be retained by the system. *)

Figure 3.6: Historical overlap at time t_3 ; both node B and node C have previously seen each other's current *write*

Input: Position of conflicting nodes in array lists from node 1 and 2, and the linked list of retained diffs.

1. $i \leftarrow$ Position in diff array of node 1
2. $j \leftarrow$ Position in diff array of node 2
- 3.
- (* *CheckHistory*: Algorithm 3, page 29 *)
4. **if** *CheckHistory*($i, j, diff1, diff2, retained$) = **true**
5. **then return**
- 6.
- (* Check if the virtual index of the collision is assigned to a cluster. *)
- (* *Cluster*: Algorithm 5, page 33 *)
7. **if** $diff1[i].cluster \geq 0$
8. **then** *Cluster*($i, j, diff1, diff2, retained$)
9. **return**
- 10.
- (* If virtual index is not a part of a cluster, resolve based on priority. *)
- (* *Priority*: Algorithm 9, page 37 *)
11. **if** $diff1[i].priority \neq diff2[j].priority$
12. **then** *Priority*($i, j, diff1, diff2, retained$)
13. **return**

```

14.
(* Resolve based on volume, if the preceding conditions do not apply. *)
(* If there is a tie in the volume, consistently break tie one way. *)
15. if diff1.length  $\geq$  diff2.length
16.   then Write(diff1[i], 1, retained)
17.   else Write(diff2[j], 2, retained)
18. diff1[i]  $\leftarrow$  null
19. diff2[j]  $\leftarrow$  null
20. return

```

The system first focuses on the priority of *writes*, then on *writes* to a specific cluster, and finally volume of *writes* from a node. The algorithms for the protocols are described below.

3.3.5.1 Cluster-based Resolution

After the system has ensured there are no historical collisions, the system calls *Cluster* (Algorithm 5, page 33). Upon instantiation of the system, indices within a cluster have already been identified as directly related or dependent upon one another. Mixing conflicting *writes* from two different nodes within the cluster could result in the data being rendered useless. Because of this, DTLRC will select *writes* to this cluster from only one node for retention in the event of a collision.

Algorithm 5

```

(* Cluster: Determines how many diffs belong to this cluster exist in both *)
(* diff lists. Resolves cluster based on priority, then volume. *)
Input: Position of conflicting nodes in array lists from node 1 and 2, and the linked
        list of retained diffs.
1.  i  $\leftarrow$  Position in diff array of node 1
2.  j  $\leftarrow$  Position in diff array of node 2
3.
(* Find other elements within either diff list that belong to this cluster. *)
(* FindCluster: Algorithm 6, page 35 *)
4.  LinkedList cluster1  $\leftarrow$  FindCluster(diff1[i].cluster, diff1)
5.  LinkedList cluster2  $\leftarrow$  FindCluster(diff2[j].cluster, diff2)
6.  int priority1  $\leftarrow$  0

```

```

7.  Diff diff  $\leftarrow$  cluster1
8.  for k  $\leftarrow$  0 to cluster1.length
9.      do
10.         if priority1 < diff.priority
11.             then priority1  $\leftarrow$  diff.priority
12.                 diff  $\leftarrow$  diff.next
13. priority2  $\leftarrow$  0
14. diff  $\leftarrow$  cluster2
15. for k  $\leftarrow$  0 to cluster2.length
16.     do
17.         if priority2 < diff.priority
18.             then priority2  $\leftarrow$  diff.priority
19.                 diff  $\leftarrow$  diff.next
20.
(* WriteCluster: Algorithm 7, page 36 *)
(* RemoveCluster: Algorithm 8, page 36 *)
21. if priority1 > priority2
22.     then WriteCluster(cluster1, diff1, 1)
23.         RemoveCluster(cluster1, diff1)
24.         RemoveCluster(cluster2, diff2)
25.     return
26. else if priority1 < priority2
27.     then WriteCluster(cluster2, diff2, 2)
28.         RemoveCluster(cluster1, diff1)
29.         RemoveCluster(cluster2, diff2)
30.     return
31.
(* If priority of writes within cluster are equal, resolve *)
(* based on volume. *)
32. if diff1.length  $\geq$  diff2.length
33.     then WriteCluster(cluster1, diff1, 1, retained)
34.         RemoveCluster(cluster1, diff1)
35.         RemoveCluster(cluster2, diff2)
36.     return
37. else
38.     then WriteCluster(cluster2, diff2, 2, retained)
39.         RemoveCluster(cluster1, diff1)
40.         RemoveCluster(cluster2, diff2)
41.     return

```

The first step within *Cluster* is to examine the *diff* lists from both nodes to determine if a node has any additional *writes* to this cluster for examination.

The system calls *FindCluster* (page 33, line 4), which attempts to locate additional *diffs* containing *writes* to the cluster in question. *FindCluster* will return a linked list of indices, which are the position within the *diff* lists of *diffs* for this cluster. *FindCluster* is guaranteed to return at least one index for *diff* lists from both nodes because the system has already determined there is a *write* collision for this index (*CollisionResolution*: Algorithm 2, page 27, line 10).

Algorithm 6

(* **FindCluster**: Iterates through the diff list and records indices *)

(* within the array that are assigned to this cluster. *)

Input: Cluster ID to search for and diff list to search through.

Output: Linked list of integers representing indices within array.

1. int *clusterID* $\leftarrow$ ID of cluster to find in the diff list
2. LinkedList *indices* $\leftarrow$ new LinkedList
3. **for** *i* $\leftarrow$ 0 **to** *diff*.length
4. **do**
5. **if** *diff*[*i*].cluster = *clusterID*
6. **then** *indices*.add(*i*)
7. **return** *indices*

Cluster then checks the *diffs* located by *FindCluster* for differences in *write* priority (page 34, line 10). Once the highest priority *write* from either cluster is located, they are compared. If the highest priority *write* from ‘node 1’ is higher than the highest priority *write* from ‘node 2’, all of ‘node 1’s’ *diffs* for this cluster are selected for retention and vice versa. If the priority is equal, the system uses the overall *write* volume from either node to determine which *writes* will be retained (page 34, line 32).

Once *writes* from one node have been selected for retention, all the *writes* to this cluster from this node are written to shared memory. The system calls *WriteCluster* (page 34, line 33) and passes in the linked list of indices, the *diff* list from the selected node, the ID for the node, and the list of retained *diffs*.

WriteCluster adds all the *diffs* for this cluster to the list of retained *diffs*, and returns.

Algorithm 7

(* **WriteCluster**: Writes the diffs belonging to this cluster to the *)

(* retained linked list. *)

Input: The list of indices of the selected diff list to write, the linked list of retained diffs, and an integer designating the origin node.

1. int *origin* $\leftarrow$ The node these diffs have originated from
2. LinkedList *index* $\leftarrow$ Linked list of indices
3. **while** *index* $\neq$ null
4. **do**
5. Write(*diff*[*index*], *origin*, *retained*)
6. *index* $\leftarrow$ *index*.next
7. **return**

Because all the *writes* to this cluster are addressed at one time, any remaining *diffs* concerning this cluster have already been examined. To ensure that the system does not reexamine the cluster during later cycles of *CollisionResolution* (Algorithm 2, page 27, line 5), those *diffs* should be removed from the *diff* list of both nodes. For this, the system uses *RemoveCluster* (Algorithm 8, page 36). *RemoveCluster* uses the list of indices created by *FindCluster* and sets any remaining *diffs* for this cluster to null.

Algorithm 8

(* **RemoveCluster**: Sets selected indices to null within the selected diff list. *)

Input: The list of indices to remove, diff list to remove indices from.

1. LinkedList *index* $\leftarrow$ Linked list of indices
2. **while** *index* $\neq$ null
3. **do**
4. *diff*[*index*] $\leftarrow$ null
5. *index* $\leftarrow$ *index*.next
6. **return**

3.3.5.2 Priority-based Resolution

If the colliding *diffs* are not assigned to a cluster, their priority is then examined. *Resolve* checks for a difference in the priority between the *diffs* (Algorithm 4, page 32, line 11). If the priority is not equal, the system calls *Priority* (Algorithm 9, page 37). *Priority* is a simple check to determine which *diff* has the higher priority. Once identified, that *diff* is selected for retention, and the colliding *diffs* are set to null.

Algorithm 9

(* **Priority**: Compares the priority of the two designated diffs, and writes *)

(* the diff with the highest priority. *)

Input: Position of conflicting nodes in array lists from node 1 and 2, and the linked list of retained diffs.

1. $i \leftarrow$ Position in diff array of node 1
2. $j \leftarrow$ Position in diff array of node 2
3. **if** $diff1[i].priority > diff2[j].priority$
4. **then** Write($diff1[i]$, 1, *retained*)
5. **else** Write($diff2[j]$, 2, *retained*)
6. $diff1[i] \leftarrow$ null
7. $diff2[j] \leftarrow$ null
8. **return**

3.3.5.3 Volume-based Resolution

Should the colliding *diffs* have an equal priority, either as a result of both having an elevated priority or the priority being left at zero, DTLRC resolves the collision based on the total volume of *writes* between either node. The rationale for volume-based resolution is that if one node has produced or is carrying more *writes* that have been created or received during the notification duration, than those *writes* should be propagated through the system when possible. The other node, having

either been less productive or had fewer contacts during its notification duration will receive new data to work with.

Resolve looks at the length of the *diff* lists from both nodes, and retains the *writes* from node with the longer *diff* list (Algorithm 4, page 31, line 15). Both *diffs* are set to null, and *Resolve* returns to *CollisionResolution* (Algorithm 2, page 27).

3.3.5.4 Remaining Diffs

CollisionResolution first iterates through the entirety of the *diff* list from ‘node 1’, and resolves any collisions identified in the *diff* list of ‘node 2’. Once collisions are resolved and *writes* selected for retention, their corresponding entry in the *diff* list of both nodes is set to null. Thusly, once *CollisionResolution* has moved through the *diff* list of ‘node 1’, any remaining *diffs* in the *diff* list of ‘node 2’ will be uncontested *writes* (Algorithm 2, page 27, line 17).

3.3.6 Applying Writes to Shared Memory

When *CollisionResolution* finishes execution, the linked list of retained *diffs* is available to share with the remote process. ‘Node 1’ submits the retained *diff* list to the remote process, followed by the *writes* from ‘node 1’ that have been selected for retention by the system. ‘Node 2’ responds with its *writes* selected for retention.

As both nodes proceed to update shared memory with newly received values, the shared memory metadata and *write* history at both nodes is also updated. *Writes* are applied to memory as they are received, with the exception of *writes* to a cluster: all *writes* to a cluster must be received at a node prior to applying them to memory so that if the two nodes are disconnected while updating memory, the cluster will not

be left in a potentially non-deterministic state. Once the update process is complete, shared memory between the two nodes is consistent.

Figure 3.7 presents a flowchart of the actions taken to resolve a collision, once the collision has been identified.

Figure 3.7: Collision Resolution Flowchart

3.3.7 Flushing Diffs

If a node is separated from its peers for the entirety of the notification duration, *writes* created by this node may ultimately be lost by the system, as the node will not generate *diffs* for these *writes* when it again encounters its peers. To address

this, a node can ‘flush’ its *diffs* if a separation exceeds the length of the notification duration.

When a node flushes its *diffs*, any *write* that was created since the last time this node encountered one of its peers will have a *diff* created for it. This ensures that should a node experience an abnormally long loss of connectivity with its peers, the data produced during that time will still be considered for retention by the system.

3.3.8 Adapting Collision Resolution Protocols

A number of settings within the collision resolution protocols can be adjusted with simple API calls in order to better suit a specific application’s needs. For instance, priority-based resolution, as presented here, will take a single *write* of priority ‘2’ from one node over multiple *writes* of priority ‘1’ from a second node. This can easily be adjusted so that the resolution protocol selects the greatest volume of elevated priority *writes* for retention, rather than a single *write* which has a higher priority than all *writes* submitted for consideration.

3.4 Social Cache

If a collection of nodes has a high probability of regularly encountering one another, they can elect to employ Social Cache. Social Cache is an expansion of DTLRC wherein a group of nodes can spread the entire contents of shared memory amongst all participating nodes. The total capacity of memory available to all systems is increased, and no individual node is storing the entire contents of shared memory locally. Social Cache is similar to how traditional distributed shared memory systems operate, while making use of the delay tolerant features of DTLRC.

In Social Cache, shared memory is divided up into indices, as in DTLRC. Each member of the system assumes responsibility for a range of indices and the contents

of memory therein. For example, in DTLRC where every node might have indices 0 to 59,999, one node in social cache would be responsible for indices 60,000 to 119,999, and another for 120,000 to 179,999 and so on, as illustrated in Figure 3.8.

Figure 3.8: Social Cache

Nodes within Social Cache are the owners of the data assigned to their range of indices. When another node wants to access that data, it first determines which node in the system is responsible for that data and enters a request into a local queue. Upon meeting the node responsible for this range of memory, a temporary copy of the data is made on the requesting node. Multiple devices can make temporary copies of a piece of data at the same time, and all can read or write to that section of memory freely.

When a node completes modifications to its temporary copy of data, it creates *diffs* and holds them in a queue. Upon meeting the owner of that data, the *diffs* are submitted and the temporary copy is dropped from the requesting node. The owner of the data maintains a notification duration based on the expected amount of time it would take to meet every node in the group. Because multiple nodes can write to a temporary copy of the data at a time, updates received from other nodes within the notification duration are passed to *CollisionResolution* (Algorithm 2, page 27), then applied to the data.

Figure 3.9 demonstrates requesting data in Social Cache. Here, node *B* is requesting data from node *A* and *C*, while node *C* is requesting data from *A*. Both *B* and *C* are requesting the same range of data, indices ‘40100 – 40200’ from *A*; All three participating can write to this range, and submit their *diffs* to *A* when their processing is complete. *A* will resolve any *write* collisions produced from those *diffs*, and the other two nodes will drop their temporary copies.

Figure 3.9: Social Cache Data Requests

As well as increasing the amount of memory available to the system, the arrangement of memory on each system no longer has to be symmetrical. A resource constrained device can allocate the amount of memory it has, while still making use of the memory on the other the other systems. This arrangement better accommodates a heterogeneous collection of computing platforms participating in the network.

At the expense in greater in memory configurations and more variety in computing platforms, a delay in requesting data and receiving data will be inherent in the operation of Social Cache. The amount of delay that can be tolerated is ultimately defined by the consumer of the data.

CHAPTER 4

SIMULATION AND ANALYSIS

This chapter discusses the simulations conducted to demonstrate the viability of DTLRC. The data traces, scenarios, plots and results are provided below.

4.1 Data Trace

The simulations used data traces from the Cambridge / Huggle dataset, provided by the Community Resource for Archiving Wireless Data at Dartmouth (CRAW-DAD) [14]. Data sets were experimentally collected by users carrying Bluetooth-enabled devices (iMotes) for several days in different social environments. The iMotes tracked sightings of other Bluetooth devices over the course of the experiment, logging the MAC address of devices they met as well as the time (relative to the beginning of the experiment) they met them.

Because the iMotes were recording any and all Bluetooth devices they encountered, the data trace distinguishes between ‘internal’ and ‘external’ contacts. An ‘internal’ contact occurs between two devices distributed by the research team; an ‘external’ contact occurs between a device provided by the research team, and a Bluetooth device not related to the experiment. This thesis uses only ‘internal’ nodes for analysis.

The iMotes scanned for Bluetooth devices every 120 seconds. Because a channel was not opened between the two nodes, when a device was spotted in two or more consecutive scans the iMotes recorded the length of that “connection” in 120 second increments. Due to this, it is not possible to determine exactly how long the devices

Table 4.1: Data Trace Details

Exp	Run Time (s)	Nodes	Contacts
1	4d 3h 46m 30s	9	1364
2	5d 6h 33m 29s	12	4229
3	2d 22h 35m 50s	41	22459
6	3d 21h 43m 38s	98	170601

were in close enough proximity to observe one another. For example, a participant might have carried their device out of a room momentarily in between scanning cycles; as long as this person returned to the room within two minutes, the data trace would still list these devices as being “connected” during that period.

Details of the experimental traces are presented in Table 4.1. The data in table 4.1 are:

- Exp: the experiment number given by the Cambridge / Haggie team. Experiment 4 is not used in this thesis and experiment 5 is not included in the released data.
- Run Time: the duration of the experiment, listed in days, hours, minutes and seconds.
- Nodes: the number devices collecting data. These are the ‘internal’ nodes.
- Contacts: the total number of ‘internal’ contacts in each data trace.

When a node contacts another node in the data trace, the length of time between these two node’s most recent and current meeting time is calculated and recorded. This ‘time last seen’ indicates how long these two nodes were separated. The maximum and median separation durations were calculated for each node, and the overall minimums, maximums and medians for all internal nodes in the data trace are presented in Tables 4.2, 4.3, 4.4 and 4.5.

Table 4.2: Exp 1 Nodal Separation Duration

	Minimum	Maximum	Median
Maximum	1h 53m 51s	3d 20h 56m 43s	2d 16h 10m 21s
Median	5m 55s	10m 14s	8m 7s

Table 4.3: Exp 2 Nodal Separation Duration

	Minimum	Maximum	Median
Maximum	1d 0h 22m 13s	2d 23h 39m 33s	2d 3h 7m 19s
Median	5m 59s	7m 55s	6m 6s

While some nodes were separated from one another for almost the whole length of their respective experiments, such a lengthy separation was abnormal. In a majority of cases, it was unusual for most nodes to go more than twenty minutes without encountering one another in these data traces.

4.2 Experiments

To demonstrate the behavior of this system in a variety of opportunistic networking environment, five categories of experiments were conducted. The setup and results of these experiments are discussed in this section.

4.2.1 Setup

Simulations conducted for this thesis demonstrate how *writes* to shared data propagate through the system. For these tests, each node shares a simulated 500KB of memory divided into 62,500 indices, or approximately 8 bytes per index. Writes

Table 4.4: Exp 3 Nodal Separation Duration, in seconds

	Minimum	Maximum	Median
Maximum	18h 34m 0s	2d 19h 13m 39s	1d 20h 44m 37s
Median	6m 1s	33m 16s	12m 8s

Table 4.5: Exp 6 Nodal Separation Duration, in seconds

	Minimum	Maximum	Median
Maximum	12h 20m 55s	3d 14h 40m 45s	2d 1h 56m 41s
Median	6m 6s	1h 1m 8s	8m 15s

to memory are uniformly distributed, and occur every five to fifteen minutes. Which nodes meet and when are defined by the Cambridge / Haggly data traces.

Three shared memory scenarios were simulated:

- One-to-one sharing: Shared memory is kept between pairs of nodes. Memory is exclusive between two nodes, and every node collaborates with every other node. Each pairing of nodes shares 500KB of memory (i.e., if node ‘A’ works with three other nodes, node ‘A’ will dedicate a total of 1500KB of memory for collaboration.)
- Group sharing: Memory is shared among randomly selected groups of four to eight nodes. The group works on one set of 500KB of memory. If the number of nodes in a group does not divide evenly into total number of nodes, remaining nodes are assigned to a partial group.
- All-to-all sharing: Every node is sharing one set of memory with every other node. All nodes in the simulation are writing to the same 500KB set of memory. Figure 4.1 illustrates the sharing scenarios.

The shared memory simulations monitored the following metrics:

- State: how many different versions of shared memory exist in the system. As processes produce *writes* to shared memory, the number of different copies of memory in the system will increase. As nodes meet and share their *writes*, the number of versions decreases as memory is made consistent. To have the same state, two nodes must have exactly the same contents of memory.
- Collisions: the number of *write* collisions that have occurred in the system.

Figure 4.1: Simulated sharing scenarios

- **Time lived:** in the event of a collision, the amount of time a *write* existed in the system prior to being overwritten. ‘Time lived’ is an indication of how long a *write* could be shared and utilized by other systems. A short ‘time lived’ would indicate that nodes are producing data that is quickly lost to the system and suggests wasted effort as these *writes* did not exist long enough to be utilized by other nodes. The longer the ‘time lived’, the longer *writes* were available to a node’s peers and suggests useful effort.

Each of the shared memory scenarios were run a total of ten times. The averages for those experiments are presented here.

4.2.2 Uncontested Write Propagation

To determine how long it would take for one *write* to propagate to every participating node in the system, an experiment was conducted to measure the spread

Table 4.6: Average Time for *Write* to Propagate to Every Node

Exp	Time (s)
1	22m 38s
2	1d 5h 53m 30s
3	17h 3m 29s
6	17h 22m 23s

of a single *write* in the absence of any *write* collisions. In each iteration of this experiment, one node from the data trace created a single *write* in an all-to-all sharing scenario. The *writes* were shared in an epidemic fashion, with no other nodes producing *writes* of their own but simply sharing the original *write* once they received it themselves. The simulation was repeated until every node was tested as the source node.

The results of the experiment are presented below.

The uncontested *write* propagation within a system is used to select a suitable notification duration for this opportunistic networking scenario. In Figures 4.4 and 4.5 for example, setting a notification duration of roughly three hours will result in 80% of nodes receiving a *write* produced by any other node. Any nodes separated from their peers for more than three hours can flush their diffs upon their return to the network (see Section 3.3.7 for details).

4.2.3 Shared Memory: One-to-One

In the one-to-one sharing scenario, each node is maintaining a discrete set of shared memory with every other node, as discussed in Section 4.2.1. Trials were repeated ten times, and the average results are presented in Table 4.7.

The minimum possible state for one-to-one shared memory is determined by the following equation, where ‘n’ is the number of nodes in the data trace: $n * \frac{n-1}{2}$. The maximum possible state is determined by: $n * (n - 1)$. In this series of experiments,

Figure 4.2: *Write* Propagation in Exp 1

the average state remained closed to the minimal possible state for this memory configuration. This indicates the rate at which nodes were meeting was high enough such that new *writes* were quickly shared between pairs of nodes and the memory state between any two working nodes was consistent more often than it was not. Few *write* collisions occurred as a result of memory being shared with as few nodes as possible.

Table 4.8 presents the ‘time lived’ for *writes* overwritten in a collision resolution protocol. Because a single set of memory was shared between a maximum of two nodes, the only way *writes* from one node could be overwritten was for the only other node to produce a *write* to the same index. Due to this, all *writes* produced in

Figure 4.3: *Write* Propagation in Exp 2

this experiment were shared with the only other available node before being dropped from the system.

4.2.4 Shared Memory: Groups

In this scenario, memory is shared between groups of four to eight nodes. The trials for each group size were repeated ten times and the averages presented here.

Referring to Section 4.2.2, the median ‘time lived’ in almost every group sharing scenario (presented in Tables 4.10 through 4.13) would fit within a notification duration set to reach 80% of nodes within the network. Because these scenarios are only sharing *writes* between a group of nodes rather than all nodes available in the

Table 4.7: Average State and Collisions for One-to-One Sharing Over Ten Trials

Exp	State	Collisions
1	48.9	3.6
2	83.9	21.0
3	849.7	65.2
6	4784.3	565.2

Table 4.8: Time Lived for Each Experiment Over Ten Trials

Group	Max	Min	Mean	Median
1	2d 23h 8m 8s	2m 13s	23h 41m 3s	16h 57m 54s
2	5d 5h 49m 15s	3m 20s	1d 10h 22m 54s	18h 53m 28s
3	3d 2h 24m 39s	1m 55s	22h 53m 35s	15h 8m 54s
6	3d 22h 9m 23s	2m 5s	22h 32m 51s	14h 47m 30s

Table 4.9: Average Collisions for Each Group Size Over Ten Trials

Nodes	Exp 1	Exp 2	Exp 3	Exp 6
4	2.2	2.8	4.9	39.3
5	4.5	10.5	10.1	79.1
6	3.2	16.8	13.4	95.7
7	6.3	7.3	15.7	118.0
8	6.2	37.3	36.6	141.3

Table 4.10: Time Lived for Each Group Size in Exp 1 Over Ten Trials

Group	Max	Min	Mean	Median
4	4d 2h 32m 43s	3m 34s	5h 26m 59s	2h 43m 21s
5	4d 1h 37m 2s	15m 44s	4h 20m 43s	1h 10m 26s
6	4d 3h 30m 37s	1m 29s	1d 8h 10m 40s	4h 45m 0s
7	4d 2h 36m 49s	4m 0s	1d 20h 52m 0s	1d 6h 48m 22s
8	4d 1h 30m 4s	5m 30s	1d 18h 39m 20s	1d 8h 16m 26s

Table 4.11: Time Lived for Each Group Size in Exp 2 Over Ten Trials

Group	Max	Min	Mean	Median
4	5d 2h 4m 9s	29m 14s	23h 17m 36s	6h 39m 4s
5	5d 5h 46m 30s	3m 19s	1d 11h 29m 30s	1d 1h 10m 55s
6	5d 6h 0m 22s	4m 13s	2d 0h 13m 11s	1d 22h 15m 50s
7	4d 3h 13m 49s	10m 8s	1d 13h 11m 5s	1d 4h 38m 27s
8	4d 3h 3m 37s	2m 34s	1d 8h 24m 58s	1d 2h 21m 48s

Figure 4.4: *Write* Propagation in Exp 3

network, it is likely that a notification duration well under the time required to reach 80% of all nodes in this environment could be utilized.

The *write* propagation results presented in Section 4.2.2 as well as the ‘time lived’ results in this set of experiments indicates that the majority of *writes* existed in the network long enough to be seen by most nodes in the group. As a majority of nodes received this *write* prior to it being overwritten at every node in the system, the *writes* that were overwritten existed with ample time to be utilized by participating processes.

Table 4.14 presents the data on state within the system. The theoretical minimum state for each group is one. The theoretical minimum state for the system is

Figure 4.5: *Write* Propagation in Exp 6

one copy of memory for each group. For example, the theoretical minimum state for groups of four within Experiment 2 is three, as four nodes in a group divides evenly into the twelve nodes available in the data trace. The maximum state for groups of four in Experiment 2 is twelve, as each node could have its own unique copy of memory.

What the state data indicates is that the rate of operations within the system is outpacing the rate of opportunistic contacts. Because DTLRC can maintain consistency between any two nodes at a time and only then when two nodes meet in an opportunistic contact, the nodes within this network are not meeting often enough to reduce the system state to its theoretical minimum value. Figure 4.6 illustrates this

Table 4.12: Time Lived for Each Group Size in Exp 3 Over Ten Trials

Group	Max	Min	Mean	Median
4	2d 2h 51m 2s	14m 13s	1d 14h 2m 1s	1d 11h 5m 22s
5	2d 1h 55m 45s	3m 58s	1d 4h 1m 40s	1d 1h 5m 48s
6	2d 2h 37m 25s	27s	1d 15h 57m 14s	1d 12h 2m 10s
7	2d 4h 21m 23s	3m 38s	1d 22h 21m 47s	1d 15h 54m 22s
8	2d 2h 31m 34s	2m 11s	1d 22h 9m 21s	1d 16h 2m 43s

Table 4.13: Time Lived for Each Group Size in Exp 6 Over Ten Trials

Group	Max	Min	Mean	Median
4	3d 20h 33m 45s	1m 46s	1d 8h 48m 44s	1d 3h 43m 0s
5	3d 12h 8m 4s	4m 34s	1d 12h 53m 53s	1d 11h 34m 49s
6	3d 13h 40m 8s	7m 34s	1d 15h 31m 49s	1d 13h 20m 59s
7	3d 17h 6m 48s	2m 23s	1d 16h 1m 1s	1d 12h 33m 30s
8	3d 20h 16m 22s	8s	1d 14h 43m 27s	1d 12h 55m 18s

phenomenon. Here, the minimum possible state in a group of five nodes is one and the maximum possible state is five. Because nodes are producing *writes* more often than they are meeting, the minimum practical state for groups of five nodes in this experiment is three. Visualized in Figure 4.6, nodes ‘A’ and ‘B’ have the same state, nodes ‘D’ and ‘E’ have the same state, and ‘C’ has a state unique from the other two groupings. In these experiments, only two nodes are sharing the exact same state at a time, because all nodes are not meeting often enough to create one consistent state before new *writes* are introduced into the system.

Table 4.14: Average State for Each Group Size Over Ten Trials

Nodes	Exp 1	Exp 2	Exp 3	Exp 6
4	6.0	8.8	29.8	65.7
5	6.9	9.0	31.5	75.7
6	6.9	9.8	33.3	76.5
7	6.8	9.6	34.1	81.6
8	6.6	9.5	33.9	81.1

Figure 4.6: Theoretical and Practical Minimum State for Groups of Five Nodes

Table 4.15: Average State and Collisions for All-to-All Sharing Over Ten Trials

Exp	State	Collisions
1	7.7	11.7
2	10.2	37.2
3	36.4	1128.4
6	88.2	31843.6

While the state data in Table 4.14 shows that groups in this experiment may seldom, if ever, obtain the theoretical minimum state, they usually operate below the maximum possible state. This indicates that nodes are meeting often enough to prevent shared memory at each device from becoming completely divergent from every other device. While reaching the minimum possible state for small groups is desirable in theory, to ensure all nodes share the exact same memory would require they reduce the rate at which they are producing *writes*, which would result in wasted time and processing potential.

4.2.5 Shared Memory: All-to-All

In the all-to-all sharing scenario, every node in the data trace is collectively operating on the same 500KB set of memory. Each trial was performed ten times and the resulting averages are presented in this section.

Table 4.15 shows but the average state and collisions for the all-to-all simulations. The theoretical minimum state for all simulations is one, and the maximum state is one unique copy of memory for each node in the system (i.e., Experiment 6

Table 4.16: Time Lived for All-to-All Sharing Over Ten Trials

Group	Max	Min	Mean	Median
1	4d 3h 38m 39s	3m 55s	2d 14h 6m 50s	2d 12h 59m 57s
2	5d 1h 36m 19s	42s	2d 0h 57m 27s	1d 20h 34m 37s
3	2d 4h 21m 23s	4m 8s	1d 23h 36m 1s	1d 19h 12m 35s
6	3d 6h 15m 15s	47s	1d 18h 57m 54s	1d 18h 36m 59s

Table 4.17: Nodes Utilized in Social Cache Simulations

Exp	Nodes
2	3, 6, 7
6	10, 13, 17

has 98 nodes, therefore the maximum possible state is 98). As discussed in Section 4.2.4, the nodes in the all-to-all simulations are not meeting often enough to agree on one set of memory before new *writes* are introduced into the system. The average state for each simulation sits well below the maximum possible state however, which shows that nodes are meeting often enough to prevent every node in the system from hosting a distinct state of shared memory.

As discussed in Section 4.2.2, the median ‘time lived’ for in the all-to-all simulation suggests that at least 80% of nodes in all data traces would have seen a *write* before it was removed from all nodes in the system. This shows that most *writes* produced at nodes are existing long enough to be utilized by a majority of processes in the system before it was dropped.

4.3 Social Cache

To simulate Social Cache, a group of nodes that met with a relatively high frequency were located in the data traces. Two sets of nodes were used and presented in Table 4.17.

Table 4.18: Social Cache Interval Between Requesting and Receiving Data Over Ten Trials

Exp	Max	Min	Mean	Median
2	16h 56m 26s	12m 36s	1h 17m 45s	46m 56s
6	16h 59m 9s	10m 21s	1h 32m 33s	12m 17s

Table 4.19: Social Cache Time Lived Over Ten Trials

Exp	Collisions	Max	Min	Mean	Median
2	1.2	3d 0h 9m 21s	10m 2s	1d 13h 34m 14s	2d 1h 14m 7s
6	2.3	3d 2h 3m 18s	3m 45s	1d 17h 55m 26s	1d 21h 57m 38s

Each node was assigned a sequential set of indices to own. The first node owned indices 0 – 62,499, the second 62,500 – 124,999 and the third 125,000 – 187,499. Which node was responsible for which range was randomly assigned, and each trial performed ten times.

Nodes requested indices with a uniform distribution. If the requested index was owned locally, the node fulfilled the request locally. If the requested index was hosted on a remote system, the node place the request for that index into a queue. When the requesting node eventually met the device responsible for that index, the request was marked ‘fulfilled’ and the interval between when the node requested the data and when the node received the data were recorded. The results are presented in Table 4.18.

The average number of collisions for each experiment per trial, as well as the data for ‘time lived’ are presented in Table 4.19. The interval between requesting and receiving data in this experiment should be viewed in the context of an application that operates on devices carried by individual people who’s mobility is primarily facilitated by moving on foot.

The average collisions in the Social Cache experiment are greatly reduced from the previous experiments utilizing DTLRC exclusively. This reflects the increased

amount of working memory available to all nodes as well as each node being responsible for its own range of memory, rather than over node operating on exactly the same set. The ‘time lived’ for these experiments indicates that most *writes* were available to all nodes participating in the system long enough to contribute to other node’s processing goals.

CHAPTER 5

DISCUSSION AND CONCLUSION

This chapter provides an example application for use with delay tolerant lazy release consistency, summarizes the contents of this thesis and presents concluding remarks.

5.1 Example Application

This section discusses an example application that could make use of the features within DTLRC. In this example, the organizers of a conference have secured discounts for attendees at restaurants near the conference venue. Menus from these eateries have been collected from the restaurant's corresponding website, and loaded into a program that can run on various smart phones. Each menu item has a quantitative rating system in place, so that participating attendees who have tried this specific dish can rate the quality. The program also allows attendees to announce events at participating restaurants to facilitate the coordination of gatherings in the conference's off hours.

Over the course of the conference, attendees can take advantage of the restaurant discounts, and rate the food they've ordered at these restaurants. As attendees encounter one another during the conference, the program can share their food reviews with others. As reviews from numerous attendees are shared amongst devices, the program can compile the ratings for each item from multiple sources, so that all attendees utilizing the program have suggestions on which restaurants and menu items are of high quality and which should be avoided.

The program would also allow a user to designate other attendees as “friends.” When a person is attempting to assemble colleagues for an off hours gathering, notifications to gatherings can be shared with everyone, or just those the users have labeled “friends.” When smart phones carried by attendees encounter one another, these invitations can be distributed to everyone, or only those specified by the user.

All of these recommendations and invitations can be exchanged amongst users with no reliance on existing networking infrastructure, such as Wi-Fi hot spots, and no usage of Internet-based servers. After the initial overhead of exchanging the available menus, all updates to the shared data utilized by this program are collaborative and completely decentralized.

5.2 DTLRC Benefits

The greatest strength of DTLRC is that it permits mobile wireless devices to attempt to collaborate with any and all wireless devices it may encounter. If two or more wireless devices share a common processing goal and have the necessary memory resources, they can begin to work together with no consideration as to when, or if, they will encounter one another again. This allows indiscriminate cooperation with a variety of peers when a device is moving through its environment. Should a device later encounter a peer, the two devices can take advantage of results produced while separated. If a device never encounters that particular peer again, no special consideration or action needs to be taken.

DTLRC can be utilized in environments where nodes have both predictable and erratic mobility patterns. Should a device shift from a predictable mobility pattern to an erratic one, no adjustments to the system need to be made. The device will continue to operate on shared memory when separated from its peers and resume cooperation when a connection is reestablished.

5.3 DTLRC Drawbacks

While DTLRC does provide a distributed shared memory paradigm for use in opportunistic network environments, it is not suitable for all applications a traditional DSM system may be used for. The inherent nature of an opportunistic network dictates that all applications used therein are delay tolerant. Because DTLRC operates with no foreknowledge of if and when a device will be encountered in the future, some DSM technologies such as data locking are not appropriate for employment over DTLRC.

Furthermore, because DTLRC requires that all the contents of shared memory be kept on each device, every node must have enough working memory available to accommodate all the shared memory contents. If two devices of disproportional resources are attempting to collaborate, the memory capacity of the entire system is reduced to a level that the device with fewer resources can make available.

5.4 Social Cache Benefits

Social Cache relieves the rigid memory configuration requirements of DTLRC and can thus accommodate more effective collaboration between robust and resource-constrained devices. Because nodes utilizing Social Cache are known to encounter one another regularly, it can be made to support some traditional DSM technologies that would be inappropriate to utilize with DTLRC.

Social Cache makes more resources available to more devices. The total capacity of memory within the system increases with every device added, and specialized resource-constrained devices such as embedded sensor systems can now participate because of the relaxed configuration requirements. One node with abundant memory and processing power in a Social Cache system can be dedicated to processing data

while other nodes are dedicated to collecting data to be processed. Social Cache also permits new memory items to be instantiated and made available because of the flexible memory configuration abilities, whereas DTLRC cannot accommodate new storage being added dynamically due to every device needing a symmetrical memory configuration.

5.5 Social Cache Drawbacks

Nodes within Social Cache assume the necessary overhead of determining if nodes will regularly encounter one another. Applications utilizing Social Cache must also consider how much delay they can tolerate, as there will be a period between when a process requests data hosted on a remote node and when the process receives that data.

Social Cache is significantly more susceptible to alterations in devices' mobility patterns. Should a node be separated from its peers for abnormally long periods of time or suffer a permanent disconnection, data hosted at that node is simply lost and numerous adjustments at the remaining nodes must be made, assuming the system can remain functional at all.

5.6 Future Work

More work can be done to develop additional collision resolution protocols. Though some resolution protocols may ultimately be application-specific, other ways to select the most appropriate *write* to apply to shared memory can be created and the existing resolution protocols enhanced.

Methods to enhance the dynamic adaptation of the system to changes in the opportunistic networking environment can be implemented. For example, reevaluat-

ing the most appropriate notification duration based on fluctuations in the rate of contacts due to changes in mobility patterns.

5.7 Conclusion

Delay tolerant lazy release consistency provides a viable distributed shared memory paradigm for deployment in opportunistic networking environments. Though not appropriate for every extant methodology used in traditional DSM systems, DTLRC does permit processes to share memory amongst one another in the absence of persistent network connections. DTLRC ensures that mobile wireless devices can continue working on shared memory even when separated from their peers for extended periods.

Social Cache extends DTLRC by permitting a greater variety of heterogeneous nodes in regular opportunistic contact to collaborate. Social Cache better accommodates collaboration between robust and resource-constrained devices and extends the total memory available for sharing at the expense of a delay between requesting and receiving data.

REFERENCES

- [1] S. Jain, K. Fall, and R. Patra, “Routing in a delay tolerant network,” *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 145 – 158, Aug. 2004.
- [2] W. Zhao, M. Ammar, and E. Zegura, “A message ferrying approach for data delivery in sparse mobile ad hoc networks,” in *Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc ’04. New York, NY, USA: ACM, 2004, pp. 187 – 198.
- [3] L. Pelusi, A. Passarella, and M. Conti, “Opportunistic networking: Data forwarding in disconnected mobile ad hoc networks,” *IEEE Commun. Mag.*, vol. 44, no. 11, pp. 134 – 141, Nov. 2006.
- [4] M. Conti and M. Kumar, “Opportunities in opportunistic computing,” *IEEE Computer*, vol. 43, no. 1, pp. 42 – 50, Jan. 2010.
- [5] W. Stallings, *Wireless Communications and Networks*, 2 ed. Upper Saddle River, NJ: Prentice Hall, 2005.
- [6] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2 ed. Upper Saddle River, NJ: Prentice Hall, 2006.
- [7] A. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms and Systems*. New York, NY: Cambridge University Press, 2008.
- [8] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, “Understanding replication in databases and distributed systems,” in *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, 2000, pp. 464 – 474.

- [9] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690 – 691, Sept. 1979.
- [10] P. W. Hutto and M. Ahamad, “Slow memory: Weakening consistency to enhance concurrency in distributed shared memories,” in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, May 1990, pp. 302 – 309.
- [11] K. Gharachorloo, D. Lenoski, J. Laudon, A. G. P. Gibbons, and J. Hennessy, “Memory consistency and event ordering in scalable shared-memory multiprocessors,” in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, May 1990, pp. 15 – 26.
- [12] P. Keleher, “Lazy release consistency for distributed shared memory,” Ph.D. dissertation, Rice University, Houston, Texas, 1994.
- [13] W.-Y. Liang, Y.-M. Hsieh, and Z.-Y. Lyu, “Design of a dynamic distributed mobile computing environment,” in *Parallel and Distributed Systems, 2007 International Conference on*, vol. 2, Dec. 2007, pp. 1–8.
- [14] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau, “CRAWDAD data set cambridge/haggle (v. 2009-05-29),” Downloaded from <http://crawdad.cs.dartmouth.edu/cambridge/haggle>, May 2009.

BIOGRAPHICAL STATEMENT

Chance R. Eary was born in Temple, Texas in 1984. He took an interest in computing at a young age when a friend told him his Nintendo was actually a computer. He went on to receive his B.S. in Computer Science from the University of Texas at Arlington in 2009, and his M.S. in Computer Science in 2011 at same. Chance has worked as a software developer with Bynari, Inc., programming message management systems, as well as working as a graduate research assistant with his supervising professor Dr. Mohan Kumar. Projects include investigating airborne networking for the Air Force Research Laboratory, and assisting with the design of a collaborative document review system for the UTA chapter of the Innocence Project of Texas (now the UTA Innocence Network). He plans to begin work on his Ph.D. in Computer Science at UTA in the Fall of 2011.