Efficient Synchronized Data Distribution Management in Distributed Simulations

A Thesis
Presented to
The Academic Faculty

by

Ivan Tacic

In Partial Fulfillment
Of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
February 2005

Efficient Synchronized Data Distribution Management in Distributed Simulations

To my parents, Ilona and Branislav, and brother Igor

ACKNOWLEDGMENTS

I am greatly indebted to Dr. Richard Fujimoto for his guidance and support throughout my doctoral study. It was under his mentoring that I developed a focus and became interested in parallel and distributed discrete event simulations. Dr. Fujimoto taught me how to do research, showing me places where I had strayed from the straight path and gently leading me back in the proper direction.

A very special thanks goes to Dr. Margaret Loper. As my supervisor and later colleague, she helped and encouraged me all along the way. I owe her my eternal gratitude for her kindness, advice and help which includes the proofreading of this thesis. I thank my thesis committee for their insightful comments and suggestions.

Huge understanding, moral and everyday support of my dear friends, Susan Craig, Kyle and Patty Crawford helped me stay focused through my ups and downs. I am very grateful that I can call them my extended family.

Finally, I would also like to thank my parents and brother Igor for believing in me and giving me their love, encouragement and support all these years. My Ph.D. would never have reached its successful conclusion without them.

CONTENTS

ACKNOWLEDGMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

SUMMARY xii

1 INTRODUCTION 1

1.1 Discrete Event Simulation 1

1.2 Parallel and Distributed Discrete Event Simulation 2

1.3 Time Management 6

1.3.1 The Synchronization Problem 7

1.3.2 Related Work 10

1.4 Data Distribution Management 11

1.4.1 The Data Distribution Problem 11

1.4.2 Related Work 13

1.4.3 Distributed Simulation Systems 14

1.4.3.1 Distributed Interactive Simulation Systems 14

1.4.4 Content-Based Routing Systems 16

1.5 Research Contributions 17

1.6 Roadmap to This Document 19

2 DATA DISTRIBUTION MANAGEMENT 20

2.1 Problem Description 20

2.1.1 Formal Problem Statement 23

2.2 Issues in Realizing an Efficient DDM System 25

2.3 High Level Architecture 28

2.4 Approaches 31

2.4.1 Region-Based Groups 31

2.4.2 Grid-Based Groups 32

2.4.3 Implications to Time Managed DDM 34

3 TIME MANAGED DATA DISTRIBUTION MANAGEMENT 37

3.1 Problem Description 38

3.1.1 Formal Problem Statement 39

3.2 Issues in Time Managed DDM 43

3.3 Synchronized DDM Approach 47

3.3.1 An Optimistic Algorithm to Time Managed DDM 49

3.3.2 Hybrid Approach to Reduce DDM Overheads: Region-Based Groups with Grids 52

3.4 Algorithm 54

3.4.1 DDM Architecture 55

3.4.2 Interest Management Layer 56

3.4.3 Distribution List Management Layer 61

3.4.4 Basic Algorithm 65

3.4.5 Correctness Proof.......72

3.5 Design Rationale....... 91

3.5.1 Dealing with Missed Messages and Messages in the Past.......91

3.5.2 Distribution List Representation.......92

3.5.3 Matching of Update and Subscriptions Regions.......95

4 PERFORMANCE EVALUATION....... 98

4.1 Implementation....... 98

4.1.1 Simplified Space-Time Memory.......98

4.1.2 Distributing the DDM Algorithm's Data Structures.......100

4.1.3 Implementing Message Resends and Retracts.......102

4.1.4 Georgia Tech's FDK Overview.......102

4.1.5 Hardware.......105

4.2 Cost of Synchronization....... 106

4.3 Benchmark Applications....... 107

4.4 Experiments....... 110

4.4.1 Impact of Lookahead on Performance.......112

4.4.2 Impact of Region Size on Performance.......113

4.4.2.1 Large Region Size....... 116

4.4.2.2 Medium Region Size....... 122

4.4.3 Impact of Network Buffers on Performance.......132

4.4.3.1 Smaller Network Buffers....... 133

4.5 Analysis 138

4.6 Performance Summary 142

5 CONCLUSIONS AND FUTURE WORK 145

5.1 Summary 145

5.2 Future Work 149

5.2.1 Precise Filtering 150

5.2.2 Utilizing Additional Multicast Groups 151

5.2.3 Automatic Self-adjusting DDM Mechanism for Improved Performance......152

APPENDIX A: FULL IMPLEMENTATION OF THE ALGORITHM 153

REFERENCES 160

LIST OF TABLES

Table 1 Large and medium region size parameters 113

Table 2 Run time confidence interval (95%) for SDDM.. 120

Table 3 Run time confidence interval (95%) for SDDM - FCS 124

Table 4 Run time confidence interval (95%) for SDDM - Synthetic 125

Table 5 Run time confidence interval (95%) for SDDM.. 134

LIST OF FIGURES

Figure 1 Two-dimensional routing space with subscription regions S_1 and S_2 and update region U. 30

Figure 2 Missed messages example. 45

Figure 3 Example of messages in the LP's past. 47

Figure 4 Message log to prevent missed messages. 50

Figure 5 DDM Organization. 56

Figure 6 Snapshot of Add and Delete operations by a single federate. The federate is subscribed to receive updates in the intervals [5,10) and [15,35). 65

Figure 7 Possible cases when a new Add or Delete operation arrives. 77

Figure 8 Data structures for federate:F0.................... 99

Figure 9 RTI architecture using RTI-Kit.................... 105

Figure 10 Run time when varying region size for SDDM. 115

Figure 11 Run time when varying region size for UDDM.................... 115

Figure 12 Run time for UDDM vs. SDDM. 119

Figure 13 Number of message sends during Updates for UDDM vs. SDDM. 120

Figure 14 DDM operations overhead for UDDM. 121

Figure 15 DDM operations overhead for SDDM. 122

Figure 16 Run time for UDDM vs. SDDM - FCS. 124

Figure 17 Run time for UDDM vs. SDDM - Synthetic. 125

Figure 18 Number of message sends during Updates for UDDM vs. SDDM - FCS.... 126

Figure 19 Number of message sends during Updates for UDDM vs. SDDM - Synthetic. 126

Figure 20 Number of message resends / retracts for UDDM vs. SDDM - FCS. 127

Figure 21 Number of message resends / retracts for UDDM vs. SDDM - Synthetic. .. 127

Figure 22 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM - FCS. 128

Figure 23 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM - Synthetic. 128

Figure 24 DDM operations overhead for UDDM - FCS.. 129

Figure 25 DDM operations overhead for SDDM - FCS. 130

Figure 26 DDM operations overhead for UDDM - Synthetic...................................... 131

Figure 27 DDM operations overhead for SDDM - Synthetic. 131

Figure 28 Run time when varying size of network buffers for SDDM........................ 132

Figure 29 Run time for UDDM vs. SDDM.. 134

Figure 30 Number of message sends during Updates for UDDM vs. SDDM. 135

Figure 31 Number of message resends / retracts for UDDM vs. SDDM..................... 135

Figure 32 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM. 136

Figure 33 DDM operations overhead for UDDM. 137

Figure 34 DDM operations overhead for SDDM... 137

SUMMARY

Data distribution management (DDM) is a mechanism to interconnect data producers and data consumers in a distributed application. Data producers provide useful data to consumers in the form of messages. For each message produced, DDM determines the set of data consumers interested in receiving the message and delivers it to those consumers. The DDM system should minimize the total number of messages that must be sent to link producers and consumers, particularly as the system scales to large number of producers and consumers.

We are particularly interested in DDM techniques for parallel and distributed discrete event simulations. Thus far, researchers have treated synchronization of events (i.e. time management) and DDM independent of each other. This research focuses on how to realize time managed DDM mechanisms. The main reason for time-managed DDM is to ensure that changes in the routing of messages from producers to consumers occur in a correct sequence. Also time managed DDM avoids non-determinism in the federation execution, which may result in non-repeatable executions. In this research we show how to efficiently synchronize DDM.

An "optimistic" approach to time managed DDM is proposed where one allows DDM events to be processed out of time stamp order, but a detection and recovery procedure is used to recover from such errors. These mechanisms are tailored to the semantics of the

DDM operations to ensure an efficient realization. A correctness proof is presented to verify the algorithm correctly synchronizes DDM events.

We propose an approach to DDM that maintains low computational and message overheads even for large scale systems. At the core of this problem is the means of matching producers and consumers of data, while keeping the number of messages that are generated at a minimum. A hybrid approach is proposed that addresses this issue for time managed as well as non-time managed DDM systems.

We have developed a fully distributed implementation of the algorithm within the framework of the Georgia Tech Federated Simulation Development Kit (FDK) software. This implementation is used to evaluate the approach. A performance evaluation of the synchronized DDM mechanism has been completed in a loosely coupled distributed system consisting of a network of workstations connected over a local area network (LAN). We compare time-managed versus unsynchronized DDM for two applications that exercise different mobility patterns: one based on a military simulation and a second utilizing a synthetic workload.

The experiments and analysis illustrate that synchronized DDM performance depends on several factors: the simulation's model (e.g. lookahead), application's mobility patterns and the network hardware (e.g. size of network buffers). Under certain mobility patterns, time-managed DDM is as efficient as unsynchronized DDM. There are also mobility patterns where time-managed DDM overheads become significant, and we show how they can be reduced.

CHAPTER 1

INTRODUCTION

1.1 Discrete Event Simulation

Simulating a system involves reproducing the behavior of a system over time. In particular, we are concerned with software simulations where the state of a system is stored in computer memory, and computation is used to change this state as time advances. At the end of a simulation, one can generate various statistics concerning the system without actually building it.

Simulation has long been an approach for analyzing systems, especially when it comes to systems where mathematical models are too complex to be solved. For example, consider an engineer who wants to design a more efficient communication network. In order to evaluate different aspects of that network, simulation can be used to find answers about network performance where classical mathematical modeling approaches are intractable. Simulations can include necessary information about the network such as topology, delay and bandwidth characteristics of the links, protocol processing performed at network nodes, etc. For a given network load, simulation can give precise information about the state of the network over time. This information can be used to identify congestion and to improve performance, e.g., by increasing the bandwidth of certain network links or processing power of nodes.

Network simulations often utilize a *discrete event simulation* paradigm. State changes in such systems are modeled by events that occur at discrete points in time. In the previous example, events might correspond to sending or receiving a message, beginning the processing of a message, etc. By contrast, in *continuous simulations* state changes are viewed as occurring continuously over time. For example, simulations used for weather forecasting typically use continuous simulation methods. The behavior of such systems are typically described with a set of differential equations. This research will focus on discrete event simulations.

A network can be modeled by a state vector characterizing the set of processing nodes that are connected via communication links. When an event representing a *message arrival* occurs, the state of a node is modified by changing the values of its variables. In our example a *message arrival* event might cause the state of the node that received the message to change to include the queuing of the message in the node’s memory buffer. It might also generate a *process message* event for the time when the message is removed from the buffer. Processing this event may cause other events to be generated.

1.2 Parallel and Distributed Discrete Event Simulation

The previous example demonstrates that one can view a physical system as a system of interacting physical processes. Each physical process has a state that is modified when certain actions occur. When the state changes, new actions either for itself or for other physical processes may occur. These actions will happen at specific points in time.

A discrete event simulation can simulate such a physical system by assigning a logical process (LP) to model each physical process. The state of an LP is the set of variables that represent the state of the physical process. Each LP includes software that models the effect of the corresponding actions in the physical system. Events are represented in the simulation by exchanging messages between LPs. Events occur at distinct points in time, and thus, messages are time-stamped accordingly. Ideally, in order to properly model the physical system, the simulation must process all events in time-stamped order. Otherwise, undesirable *causality errors* may occur where future events affect those in the past. *Synchronization* of events is needed to ensure such causality errors do not occur.

Parallel and distributed discrete event simulation refers to the execution of a discrete event simulation program over multiple processors. *Parallel* discrete event simulations execute on tightly coupled computer systems such as shared memory multiprocessors. All processors are interconnected via high-speed switches, and hence, communication latencies are low. On the other hand, *distributed* discrete event simulations are executed on loosely coupled computer platforms, such as workstations interconnected via commercial local or wide area networks (LANs and WANs). Communication latencies are substantially higher than those for tightly coupled systems, and are usually at least an order of magnitude higher on distributed computing platforms.

Depending on their use, parallel and distributed discrete event simulations can be classified into two categories:

- *Analytical Simulations.* Simulations used to analyze systems must mimic the causal relationships (i.e. before and after relationships) of a physical system precisely or as closely as possible. They are typically used to obtain accurate statistics about the behavior of the physical system being modeled. Hence, one of the key characteristics of analytical simulations is repeatability, that is, the simulation should always yield the same outputs if the same input parameters and initial LPs' states are used. Also, these simulations typically run in an as-fast-as-possible manner, in order to complete the simulation run as quickly as possible. Finally, analytical simulations often run without any user interactions, except for the fact that a user is allowed to observe the outputs and state of the modeled physical system during the simulation execution. An example of such a simulation is modeling a telecommunication network. A designer uses the simulation in an iterative manner to evaluate and verify a network design. She/he may be interested in understanding how the network performs under different conditions, and revise the network design to maximize the performance or reliability.
- *Distributed Virtual Environment Simulations.* This is a second category of simulation applications. These simulations are used to create a virtual world, e.g., for training or entertainment applications, that appear realistic to its participants, to meet the objective of the exercise. Before and after relationship may not always be perceptible by human participants, so casual relationships can sometimes be

relaxed. Typically, these simulations run in real-time since humans are actively participating, e.g. controlling the behavior of entities in the model. The required degree of realism depends on the purpose of the simulation.

Analytical and distributed virtual environment simulations often require different levels of accuracy, as was seen from the above examples. This leads to different requirements with respect to the ordering of events in a parallel or distributed discrete event simulation. Consequently, different synchronization mechanisms are typically used for these categories of simulations. Synchronization algorithms for analytical simulations will be discussed later.

This research focuses on two aspects of parallel and distributed discrete event simulations:

1. Every simulation can be viewed as a collection of LPs exchanging messages. The idea behind *data distribution* is to interconnect LPs which produce data with LPs that need this data in a simulation. For each message produced at an LP, the simulation executive must determine the set of data consumer LPs interested in receiving the message and deliver it to those consumers. This topic is known as Data Distribution Management.
2. Data distribution events, like all other events must be synchronized for analytical simulations. In other words, all events, including data distribution events must be processed in time stamp order at each LP. Synchronization of data distribution events is referred to as Time-Managed Data Distribution.

1.3 Time Management

The execution of a distributed simulation involves at least three distinct notions of time [4]:

1. *Physical time.* This time refers to time in the physical system, i.e. the actual system being modeled. Recall that the physical system can be viewed as a collection of interacting physical processes, where the state of the system evolves over physical time.
2. *Simulation time.* This time is the representation of the physical time in the simulation. We use simulation time to assign time-stamps to events during the execution of the simulation. In a discrete event simulation state changes occur at discrete points in simulation time.
3. *Wallclock time.* This time refers to the time during which the simulation is executing on a computer platform. It is the time obtained by reading the computer's real-time clock.

In simulations executing in as fast as possible mode, there is usually no direct relationship between wallclock time and simulation time. On the other hand, advances in wallclock and simulation time are typically paced in distributed virtual environment simulations.

Processing events from all LPs in time-stamp order guarantees that no causality errors occur. This is easily accomplished in a sequential simulation. One need only maintain a list of all unprocessed events in the system, and process them in non-decreasing time-stamp order. The simulator removes the event with the smallest time stamp from the list, and processes that event. Processing an event includes performing a computation that models the behavior of the corresponding physical process when that event occurs. Thus, processing an event typically results in changing the state of the LP. As a result, one or more new events may be generated for other LPs. These new events are inserted into the event list.

In a parallel or distributed simulation the execution of the LPs may be distributed across different CPUs. At first glance, the original simulation paradigm where LPs exchange messages fits this mode of execution very well. Events are processed in time-stamp order in each LP, and events generated for other LPs are sent to CPUs containing those LPs. If all events, i.e. both local events and those received from other LPs, are processed at each LP in non-decreasing time stamp order, then this mode of execution yields the same result as the sequential simulation.

1.3.1 The Synchronization Problem

Some mechanism is required to ensure that each LP processes events in non-decreasing time-stamp order. Without such a mechanism, nothing prevents a situation where an LP processes an event from the event list, and later receives an event with a time-stamp smaller than the one it has already processed. Events need to be synchronized to ensure this does not happen. In particular, this research focuses on the synchronization of data distribution events. Data distribution is presented in section 1.4.

But, which events need to be synchronized and which events can be processed concurrently? If two events affect the same state, they must be synchronized. For example, two events on the same LP that modify the same portion of the LP's state cannot be executed concurrently. Events on different LPs may also have to be synchronized, since processing an event at an LP may generate an event for another LP, and hence indirectly affect the state of another LP. Generally, it is difficult to know the events that will be generated during the simulation execution and what LPs' states they will affect. However, it suffices for each LP to process events in time-stamp order, in order to guarantee the same results are produced as in the sequential execution.

Clearly, the partitioning of the physical system into physical processes which are mapped to logical processes determines how much concurrent execution can be achieved. This is an important task in modeling the system, but will not be discussed here. Rather, we concentrate on synchronization of the system once the LPs have been defined. Generally, there are two approaches to synchronization termed *conservative* and *optimistic* discrete event simulation.

A conservative simulation ensures no event will be processed by an LP until it can guarantee that no event with a smaller time stamp will later be received by that LP. Because an event could cause a message to be sent to every other LP with the same time stamp, this could lead to very poor performance. To overcome this problem, the *lookahead* is introduced. One approach is to assign a lookahead value to each LP. When an event is being processed, all new events generated as the result of processing that event must be at least that LP's lookahead value in the future. For example, processing an event with time-stamp T at an LP with lookahead L can only result in new events with a time-stamp greater than or equal to T + L.

How does lookahead help in the concurrent processing of events? Consider a situation where all LPs have the same lookahead. If T is the smallest time-stamped event in the entire system, then we can safely process events with time stamps less than T + L and guarantee no LP processes events out of time stamp order.

Optimistic simulation is an alternative approach to synchronization. While lookahead is a straightforward concept for concurrent processing, it sometimes imposes difficult constraints on the model. In addition, the lookahead may be too small to achieve acceptable levels of parallelism for some systems. For these reasons, many have examined optimistic synchronization methods, where an event may be processed despite the fact that later an event with smaller time-stamp may arrive. To insure the simulation's final state is the same as that obtained in a sequential simulation, state saving and rollbacks are introduced. When an LP receives an event in its past, it must roll back to a previously saved state. In addition, if there were any events or messages generated for other LPs by rolled back events, they also must be cancelled. This is accomplished using an anti-message mechanism [37]. When an LP receives an anti-message, it will also have to roll back to one of its previously saved states, and generate anti-messages if the cancelled event has already been processed.

One issue in optimistic simulation is to know how optimistically LPs can execute, or in other words, how far LPs can advance ahead of each others during the execution. It may happen that advancing too far ahead results in too many rollbacks, resulting in much wasted computation. Secondly, there is a need for efficient state saving and rollback techniques.

1.3.2 Related Work

A detailed overview and discussion of both conservative and optimistic techniques can be found in [4]. The null messages algorithm by Chandy and Misra [38] is one of the first conservative approaches. Null messages are used to provide other LPs with a lower bound on the time stamp (LBTS) of future messages sent from one LP to another. Too many null messages may degrade the performance of such systems, and later approaches addressed this issue. They compute the LBTS by taking into account the time of the next unprocessed event and its lookahead, allowing LPs to advance simulation time faster [40, 47, 48].

The first optimistic synchronization approach based on rollback and anti-messages was Jefferson's Time Warp mechanism [37]. Later approaches focus on optimizing Time Warp's aggressive optimistic execution. For example, lazy cancellation [49] delays sending anti-messages until the reexecution of the computation caused by a rollback determines the reexecution did not generate the original messages again. On the other hand, lazy reevaluation [50] avoids rolling back events when a message in the past does not change the LP's state. Other techniques, such as [51] attempt to constrain the amount of optimistic computation, that is computations that may later need to be rolled back. [52] explores further how to add optimism to existing conservative simulation mechanisms. Several other mechanisms limiting optimism have been proposed in the literature [4].

1.4 Data Distribution Management

Consider a simulated virtual battlefield where objects represent entities such as tanks, airplanes, artillery, infantry, etc. At any time during the exercise, each of these objects can only “see” a subset of all the other objects in the simulated battlefield. Tanks may, for example, see other objects within the range of its sensors, while infantry units may interact with other units that are tuned to the same radio frequency. The exercise progresses as objects modify their current state according to the other objects with which they may interact.

1.4.1 The Data Distribution Problem

The simulated game scenario presents a problem known as Data Distribution Management (DDM). As mentioned above, an object can interact with other objects based on some criteria. Consumer objects must subscribe to receive messages from objects they must track, e.g., a tank object might subscribe to the position attribute of other tanks in the range of its sensors. Data must be routed between data producers and data consumers. Data producers provide useful data to consumers in the form of messages. For each message produced, DDM must determine the set of data consumers interested in receiving the message and deliver it to those consumers.

In order for DDM to determine the routing of data between producers and interested consumers, there must be an agreement between producers and consumers concerning how consumers express their interest in data, and how producers label messages that are being sent. There must be a language for expressing interests and labeling messages. The next chapter will provide a general framework for such a language. This framework is applicable not only in the simulation domain, but also in other application domains using data producers and data consumers.

Data Distribution Management is important for simulation applications in order to:

- Achieve *smaller communication latencies* by reducing the amount of required communication; reduced communications also *reduces computational requirements* for sending and receiving messages.
- *Speed up* a simulation execution by reducing the overhead necessary to implement synchronization algorithms.

The benefit of achieving smaller communication latencies is particularly important in Distributed Virtual Environment applications. An efficient DDM mechanism is essential to realize scalable systems.

DDM helps in achieving smaller communication latencies between LPs as well as in reducing computational requirements. However, an efficient implementation must be realized. Broadcast is one example of a simple, but inefficient DDM implementation. We may experience two problems. One is the amount of unnecessary messages flowing between data producers and data consumers, making the network bandwidth requirements excessively large. Second, unnecessary messages cause processing delays because data consumers must filter these unwanted messages.

1.4.2 Related Work

DDM has appeared in the literature in different domains. We will describe work in the context of two different domains: distributed simulation and content-based routing systems.

In order for DDM to determine the set of data consumers interested in receiving the message, there must be a common vocabulary for expressing interests and labeling messages between data consumers and data producers. This vocabulary can be viewed as a set of values, and expressing interest or labeling a message always specifies a subset of the vocabulary. DDM systems can be categorized based on the cardinality (i.e. size) of the vocabulary and the complexity in expressing interests and labeling messages.

First generation DDM systems used a finite vocabulary. Elements of the vocabulary can be viewed as the subjects or groups, usually statically defined, in which consumers may express their interest. For example, in a newsgroup application, we have a finite number of newsgroup names to which participants are allowed to subscribe (i.e. express an interest), or post messages (i.e. label a message). The DDM vocabulary consists of the newsgroup names, while expressing an interest or labeling a message is done by specifying a subset of newsgroup names. Hence the name *subject-based DDM systems.*

Second generation DDM systems use an infinite vocabulary. These systems are commonly known as the *value-based* or *content-based DDM systems*. The idea behind value-based DDM is to have the ability to express interest not only in one or even a few predefined subjects, but also in the range or multiple ranges of subjects (or values) that can be uncountable. Therefore, the vocabulary also has an infinite set of values. For example, in a stock-trading application the vocabulary may consist of (stock name, price) tuples, where stock name is a string and price a real number. Expressing an interest such as (wheat, (price < 100.00)) specifies an uncountable range, since the price is a real number. In other words, we want to be able to express interests based on the values computed during the execution.

1.4.3 Distributed Simulation Systems

1.4.3.1 Distributed Interactive Simulation Systems

Distributed Interactive Simulation (DIS) is a set of standards for creating realistic and highly complex distributed virtual environments that may include different simulators such as (1) human-in-the-loop simulators for modeling individual platforms (e.g. tank simulators), (2) aggregate simulators for modeling groups of units (e.g. war gaming simulators), and (3) live elements that model embedded physical components (e.g. instrumented missiles). DIS seeks to provide interoperability between different simulators.

Data distribution management mechanisms must provide efficient, scalable support for large-scale distributed simulations. They have been extensively studied in distributed simulations for training. Work in distributed simulation environments in the SIMNET (SIMulator NETworking) project [21] and many DIS systems broadcast each state update (event) to all simulators in the exercise. It is well known that this approach does not scale because the amount of communications is $O(N^2)$ where N is the number of processors. CPUs become overloaded processing incoming messages, most of which are discarded (for large N) because they are not relevant to the entities simulated within the processor. Further, communication bandwidth requirements become excessively large as N increases. It has been estimated that 375 MBits per second per platform would be required for a simulation exercise including 100,000 players [22].

Several approaches to attacking this problem have been proposed (see [23] for a survey on this subject). Virtually all use some mechanism to send messages only to the destinations that have a need for the information rather than broadcast it to everyone. For example, in the Joint Precision Strike Demonstration (JPSD)[24], federates (e.g. LPs) indicate what information they wish to receive by specifying predicates on entity attributes. Many simulators, e.g., ModSAF[25], CCTT[26], and NPSNET[22] use grid cells to filter information. A two-level hierarchical filtering scheme used in an optimistic parallel simulation is described in [27], and a generalization of grid cells using a construct called routing spaces is used in STOW[28].

The focus of the work described here is on the routing space approach that has been incorporated into the baseline definition of the High Level Architecture (HLA), though many of the techniques and mechanisms are applicable in other contexts. HLA is in fact a successor of DIS, and it is an architecture intended to provide integration of both distributed virtual environment simulations and analytical simulations. Chapter 2 describes HLA and the routing space concept in detail.

1.4.4 Content-Based Routing Systems

Work in the Gryphon project at IBM [12] is one example of a content-based distribution system. In a simple stock trading application the vocabulary may consist of a tuple (issue, price, volume), where issue is an enumerated type with 100 choices, one of which is IBM, while price and volume are real numbers ranging from 0 to 1000. A valid tuple is thus (IBM, 100, 500). Expressing an interest may take the form: (issue = IBM) $\cap$ (price < 100). The system ensures the timely delivery of published events to all interested subscribers over a logical network of brokers. The assumption is that brokers know the topology of the network. A centralized matching algorithm is transformed into a distributed version, where each broker, upon receiving a message, sends a point-to-point message only to brokers along a path to interested subscribers. Multicast services at the network level are not exploited. A similar approach is employed in SIENA [14].

More recent work in the Gryphon project [13] introduced several ways to define multicast groups. One of the proposed approaches requires a large number of groups in large-scale systems. An alternate approach using heuristics to form groups was developed to solve this problem, at the expense of introducing additional computation overheads.

Systems such as Yeast [15] and Elvin [16] are other examples of systems supporting rich ways to express interests. In Elvin, the vocabulary consists of some number of enumerated and typed data elements (e.g. integer, floating point, string data types, etc.). To label a message a set of such enumerated and typed data elements is defined. Expressing an interest is in fact a boolean expression over the elements of the vocabulary. Multicast is not utilized in these systems. These systems use a client server architecture where there is a single server to which all events are first sent, and the events are then forwarded to subscribers of only those events for which the client has expressed interest. In Elvin, using "quench" function, some events are not sent to the server at all if it is determined there are no subscribers for those events. Yeast is a general-purpose platform for building distributed applications in an event-based architectural style. It is tightly integrated to the UNIX system, allowing the monitoring and definition of rules in terms of operating system events related to the file system, groups and users.

There are other algorithms that exploit IP multicast, which are not based on the producer-consumer paradigm, but are informative due to the way groups are formed. Web caching systems such as LSAM [17] and Adaptive Web Caching [18] use IP multicast, however they do not consider multicast groups to be a limited resource. The idea in LSAM is to reduce the first-hit cost of page retrieval throughout the system by using multicast to distribute web pages to a set of proxy caches emulating a single, central shared proxy cache. Groups are formed based on predictions concerning likely interest in particular pages. Adaptive Web Caching maintains a mesh of overlapping multicast groups, over which cache trees are implicitly formed. Overlaps allow frequently visited pages to propagate closer to end clients over time. Only a few caches near the origin server, on the other hand, will see pages with infrequent requests.

1.5 Research Contributions

The research described here is concerned with the realization of efficient data distribution mechanisms for distributed simulation applications. The specific contributions of this research are concerned with *Time managed DDM*. Previously, most research has treated time management and DDM independent of each other. This research focuses on how to realize time managed DDM mechanisms.

- *Time managed DDM algorithm*. Our time managed DDM algorithm is optimistic in that it allows events to be processed out of time stamp order, and an error detection and recovery procedure is used rather than strictly avoiding errors. Furthermore, it is tailored to the semantics of the DDM services, thereby avoiding the use of general rollback mechanisms, which are more complex and require large runtime overheads.
- *Correctness proof*. Besides proving the correctness of the time managed DDM algorithm, our proofs also highlight the important properties of the algorithm.
- *Distributed implementation*. We have developed a fully distributed implementation of the algorithm within the framework of the Georgia Tech's FDK software. FDK is designed to facilitate building efficient run-time infrastructures that are necessary to federate simulations.
- *Performance evaluation of time managed DDM overheads*. A performance evaluation of synchronized DDM has been completed in a loosely coupled distributed system consisting of a network of workstations connected over a 100 Mbps LAN. Experiments indicate that the synchronized DDM system can perform as efficiently as an unsynchronized DDM system in this computing environment for some, but not all situations.
- *Reduction of DDM overheads*. An initial DDM implementations had large computational overheads and/or generated large number of messages. Hence, the

simulations using DDM couldn't scale very well. We present an approach that reduces computational and message overheads at the same time. The hybrid approach proposed here reduces DDM overheads effectively for time managed as well as non-time managed DDM systems.

1.6 Roadmap to This Document

The rest of the document is organized as follows. Chapter 2 describes the DDM problem and the issues in realizing efficient DDM systems. This is followed by explaining the routing space framework for DDM that has been incorporated into the baseline definition of the High Level Architecture (HLA), intended for federating simulation systems. The merits and drawbacks of other DDM approaches are then discussed.

Chapter 3 describes our optimistic approach to time managed DDM. First we describe the problem, followed by the synchronization issues that must be addressed. Then we explain the optimistic approach to synchronization in section 3.3.1, and the hybrid approach to reduce DDM overheads in section 3.3.2. Section 3.4 presents our time managed DDM algorithm and correctness proofs.

Chapter 4 describes the implementation within the framework of Georgia Tech's FDK software and performance results. Finally, Chapter 5 summarizes the contributions of this thesis along with the directions for future work.

CHAPTER 2

DATA DISTRIBUTION MANAGEMENT

2.1 Problem Description

Data distribution management (DDM) is an approach to interconnect applications in a distributed computing environment. In this section we first present the basic framework common to any DDM system, followed by a formal problem description. Then the issues in realizing an efficient DDM system are explained.

The objective of the DDM system is to interconnect data producers and data consumers in a distributed application. Data producers provide useful data to consumers in the form of messages. For each message produced, the DDM system must determine the set of data consumers interested in the message and deliver it to those consumers. To be able to do this, producers and consumers must agree on how to describe information contained in the messages that are being produced and how to express interest in messages based on their content.

In general, a DDM system can be viewed within a basic framework that includes a *name space* and a language to specify *expressions* to delineate portions of the name space [4]. Producers specify information they are generating via *description* expressions, while consumers specify information they wish to receive via *interest expressions*. When the description expression associated with a message overlaps with an interest expression associated with a consumer, the message is sent to that consumer. To illustrate this, consider a newsgroup system with three newsgroups: *cooking, travel* and *sports*. The set of newsgroup names defines a name space, while description and interest expressions of the language are subsets of this name space. For example, a message associated with a description expression containing *cooking* will be sent to a user whose interest expression contains *cooking* and *travel*, but not to one whose interest expression only contains *travel and sports*.

More formally, the basic DDM framework consists of the following elements:

Name space. The name space is the set of all possible values to which interest and description expressions may map. The name space is a set of tuples $V = (V_1, V_2, \ldots, V_N)$, where each V_i is a value of some basic type, called an attribute, or another tuple. For example, in a simple stock trading application the name space may consist of a tuple (issue, price, volume), where issue is an enumerated type from 100 choices, one of which is IBM, while price and volume are real numbers ranging from 0 to 1000. A valid tuple is thus (IBM, 100, 500).

Language and Expressions. The language consists of all possible expressions to specify a portion of the name space. An expression is a single test or a conjunction of tests of an arbitrarily complex function with some or all of its arguments in the name space:

Expression $E := \rho_1(F_1(v), v_1) \cap \rho_2(F_2(v), v_2) \cap \ldots \cap \rho_K(F_K(v), v_K)$

That is, it is a single relation or an intersection of relations of type $\rho(F(v), v_T)$ for a given test point in the name space $v_T \in V$, where $F(V) \rightarrow V$ is an arbitrary function with its codomain being the name space, while F's domain can, in general, include arguments not related to the name space (e.g. state variables of the simulation, etc.). $\rho(F(V), V)$ is an arbitrary binary relation used to test the value of function F. Note that v_T may be specified by not listing the values for all the attributes in the name space. In case one attribute is missing it represents a line segment in the name space, or when two or three attributes are missing it represents an area or a volume in the name space respectively, etc. Hence, we say v_T represents a hyper-point in the name space.

The expression evaluation V(E) is defined as $\{ v \mid v \in V$ and $\rho_1(F_1(v), v_1)$ and $\rho_2(F_2(v), v_2)$ and… and $\rho_K(F_K(v), v_K) \}$, that is, the expression evaluates to a set of points in the name space where each tuple satisfies all of the binary relations.

Interest expressions. When consumers use expressions they are called interest expressions. They are used to specify what information is of interest to be received. Using the previous example, an interest expression for a language for the name space may be: (issue = IBM) $\cap$ (price < 100). The first relation is the "=" relation and the second is the "<" relation. The functions used are identity functions, i.e. $F_1 (x) = x$ and $F_2 (x) = x$, while IBM and 100 are hyper-points in the name space. This expression maps to a portion of the name space, which can visually be represented as a rectangular area in this 3-dimensional name space.

Description expressions. When used by information producers expressions are called description expressions. A description expression is associated with each message. For example, the description expression (issue = IBM) $\cap$ (price = 90) should cause a message to be sent to a consumer whose interest expression is given above. However, a message associated with a description expression (issue = COCA COLA) should not be sent to this consumer.

2.1.1 Formal Problem Statement

Definitions

- Let $\boldsymbol{P}$ represent a set of all processes in a distributed application: $\boldsymbol{P} = \{ P_1, P_2, \dots , P_{Np}\}$ where Np is the total number of processes
- Let $\boldsymbol{E}$ represent a set of all expressions in a distributed application: $\boldsymbol{E} = \{ e_1, e_2, \dots , e_{Ne}\}$ where Ne is the total number of expressions
- Let $\boldsymbol{D}$ be a set of all description expressions such that $\boldsymbol{D} \subseteq \boldsymbol{E}$
- Let $\boldsymbol{I}$ be a set of all interest expressions such that $\boldsymbol{I} \subseteq \boldsymbol{E}$
- Let $\boldsymbol{p}$ be a mapping $\boldsymbol{p}$: $\boldsymbol{E} \rightarrow \boldsymbol{P}$ designating a process that produced an expression. For example, if process P_j produced expression e_i we can write $\boldsymbol{p}(e_i) = P_j$
- Let $\boldsymbol{r}$ be a mapping $\boldsymbol{r}$: $\boldsymbol{D} \rightarrow \boldsymbol{P}$ designating all consumer processes (i.e. recipients) whose interest expressions overlap a description expression. That is, for a description

expression d $\in$ ***D***, P $\in$ ***r*** (d) if and only if $\exists$ i $\in$ ***I*** such that ***p***(i) = P and V(d) $\cap$ V(i) $\neq \varnothing$

- Let a multicast group $\boldsymbol{G_i}$ be a subset of all processes, that is $\boldsymbol{G_i} \subseteq \boldsymbol{P}$. Three operations can be performed on a multicast group:

    1. The ***Mcast_join_group(G***$\boldsymbol{_i}$***, P***$\boldsymbol{_j}$***)*** operation adds process $\boldsymbol{P_j} \in \boldsymbol{P}$ to the multicast group $\boldsymbol{G_i}$.
    2. The ***Mcast_leave_group (***$\boldsymbol{G_i}$***,*** $\boldsymbol{P_j}$***)*** operation removes process $\boldsymbol{P_j} \in \boldsymbol{P}$ from the multicast group $\boldsymbol{G_i}$.
    3. The ***Mcast_message (***$\boldsymbol{G_i}$***, msg_content)*** operation sends a message with content ***msg_content*** to all processes in $\boldsymbol{G_i}$. This is also referred to as multicasting a message to the group. ***msg_content*** contains all the necessary information about the message content, e.g., the memory location where the message content starts and the length of the message.

DDM problem statement

For a given number of multicast groups $\boldsymbol{Ng}$ and $\boldsymbol{P}$, $\boldsymbol{E}$, $\boldsymbol{D}$, $\boldsymbol{I}$, $\boldsymbol{p}$ and $\boldsymbol{r}$ find:

1. A set of multicast groups $\boldsymbol{G} = \{ G_1, G_2, \dots , G_{\boldsymbol{Ng}}\}$ where $G_i \subseteq \boldsymbol{P}$

   and

2. A mapping $\boldsymbol{m}: \boldsymbol{D} \to \boldsymbol{G}$ such that $\forall$ d $\in \boldsymbol{D}$, $\boldsymbol{r}(d) \subseteq \bigcup_{\forall G_i \in \boldsymbol{m}(d)} G_i$

2.2 Issues in Realizing an Efficient DDM System

Here, a multicast group refers to a set of destination processes. We do not consider how group communications are implemented, which may have a large effect on performance and memory requirements. For example a multicast group may be implemented with a native multicast group communication facility provided by network services, or it may be implemented using point-to-point communications. The latter is clearly a less efficient implementation. We assume reliable communication, i.e., every message sent to a group is eventually delivered to each process that is a member of the group when the message is sent.

As the previous section indicates, a central problem in realizing DDM concerns the definition and composition of the multicast groups. Interest expressions must be mapped to groups to which the consumers must join. Description expressions associated with a message are mapped to one or more groups to which the message must be sent.

The mapping of description expressions to groups must be such that each message will be routed to all interested consumers. Ideally, a consumer should only receive messages for which it has expressed an interest. This is, unfortunately, not always practical due to limitations in the number of multicast groups that the network may support and the computational complexity necessary to determine optimal mappings that produce the minimal number of messages. Hence, several issues concerning the definition of multicast groups and the mapping of expressions to the groups must be addressed by DDM.

Duplicate messages may occur. For example, if a description expression is mapped to two groups that both include the same consumer, two identical copies of the message will be sent to the consumer over these groups. This must be addressed, e.g., by filtering the extra message at the consumer.

Extra messages may occur. When a multicast group mapped to a description expression contains a consumer not interested in messages associated with this description expression, the consumer will receive each such message. These extra messages must be avoided, e.g., by filtering the message at the consumer.

Control messages may be needed. These are messages exchanged during group modifications (e.g. join and leave operations on a multicast group). One can envision an implementation where interest and description expressions are provided at the application initialization phase and never change. In this case no additional control messages are required after the multicast groups are formed. Another implementation involves changing the expressions during application execution. The number of control messages necessary to reconfigure multicast groups varies depending upon the chosen mapping and implementation of groups.

One way to realize DDM is using point-to-point communication (i.e. unicast). In other words, all consumers for each description expression must be determined, and then messages associated with that expression must be sent to each consumer via a point-to-point message send. This causes delays at the producer, and may cause more network congestion than is necessary. This implementation can avoid duplicate, extra and control messages, however. It is worth noting that this approach works well if groups are small.

At the other extreme, broadcast communication may be used. Broadcast involves sending each message only once, which is then received by all consumers regardless of their interest. Filtering of such extra, uninteresting messages must be performed at the consumers. A situation may result where the consumers are filtering and discarding most of the messages they receive. Furthermore, the amount of communication becomes excessive as the number of participants increases. For N participants (e.g. processors) the amount of communication is $O(N^2)$, so this approach does not scale.

Multicast network services are an attractive alternative to realizing efficient DDM, which promises better scalability. Ideally, an efficient DDM system would easily be realized if there were enough multicast groups available (e.g. at least as many as description expressions). A multicast group could be assigned to each description expression. Changes in description or interest expressions may cause group membership modifications. The advantage of this approach is that sending a message to an appropriate multicast group uses less of the network bandwidth than broadcasting the message. However, control messages that occur during join and leave operations introduce overheads, which must be balanced with the benefits of using multicast groups for message sends.

2.3 High Level Architecture

The focus on this research is on the routing space approach that has been incorporated into the baseline definition of the High Level Architecture (HLA), though many of the techniques and mechanisms are applicable in other contexts. HLA is in fact a successor of DIS, and it is an architecture intended to provide integration of both distributed virtual environment and analytical simulations.

DDM takes on a somewhat different flavor in distributed simulations constructed by “federating” existing simulators compared to a traditional parallel discrete event simulation (PDES) program. PDES programs typically assume each logical process (LP) is responsible for determining which other LPs should receive the messages it generates. By contrast, federated simulation systems typically implement this functionality in the underlying distributed simulation software, referred to as the Run-Time Infrastructure (RTI) in High Level Architecture (HLA) terminology, rather than within the simulation model. Each simulator (federate) specifies via interest expressions what messages are of interest.

An example of this approach is class-based subscription used in the HLA. The name space consists of (class, attribute) tuples where the class hierarchy is known prior to execution. Besides defining its own attributes, a class inherits all the attributes from its parent class. For instance, a class called *vehicle* may be defined, with subclasses *aircraft, tank* and *truck.* An attribute *position* is defined in the *vehicle* class, and inherited by all three subclasses mentioned. Thus the name space consists of the tuples *(vehicle, position), (aircraft, position), (tank, position), (truck, position).* Expressing interest in an attribute in the class hierarchy tree causes a corresponding interest expression's evaluation to include name space tuples with the same attribute of all the classes in the subtree rooted at that class. A federate's (i.e. a simulation's) interest expression may be (F_1 (class, attribute) = *(vehicle, position)*), where the relation "=", function F_1 and test point *(vehicle, position)* are used to construct an expression. Function F_1 is more complex than in the newsgroup example from the beginning of this chapter, where it is a simple identity function $F(x) = x$. For a given test point *(vehicle, position)* in the name space this expression results in carving the portion of the name space so that it will include all tuples (class, *position*), with class being an original class (*vehicle* in this case) and all the classes in the subtree rooted at the original. Thus, the resulting portion of the routing space described by this interest expression is *(vehicle, position), (aircraft, position), (tank, position), (truck, position).* Description expressions, on the other hand use only the identity function F_2. For instance, (F_2 (class, attribute) = *(tank, position)*) is a description expression evaluating to a single tuple *(tank, position)*. A federate interested in position updates for all vehicles, or a tanks' position update only, will receive messages associated with this description expression. Those with test points *(aircraft, position)* and *(truck, position)* will not receive such messages.

DDM using the *routing spaces* concept for distributed simulations is an example of value-based DDM. Data Distribution Management services in the HLA are used to specify the routing of data among federates. In the HLA, value-based DDM is based on an n-dimensional coordinate system called a routing space. For example, a two-dimensional routing space might represent the play box in a virtual environment. A rectangular update region can be associated with each update message generated by a federate. Federates express interests via rectangular subscription extents. Multiple extents form a region. If the update region associated with a message overlaps with a federate's subscription region, the message is routed to the subscribing federate. For example, in Figure 1 updates using update region U are routed to federates subscribing to region S_1 but not to federates subscribing to region S_2.

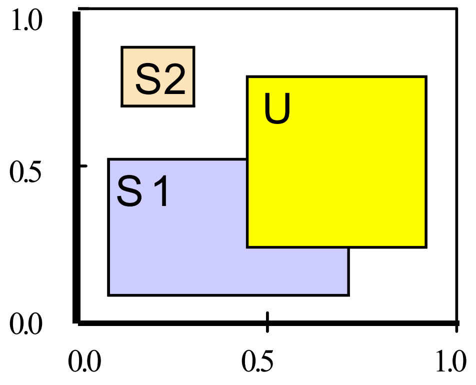


Figure 1 Two-dimensional routing space with subscription regions S_1 and S_2 and update region U.

The name space for an n-dimensional routing space is a tuple (X_1, X_2, … X_N) with $X_{MIN} \leq X_i \leq X_{MAX}$, where X_{MIN} and X_{MAX} are federation-defined values. For example, Figure 1 shows a two-dimensional routing space with axis values ranging from 0.0 to 1.0.

Update regions are in fact description expressions, while subscription regions are interest expressions. As mentioned earlier, regions are collections of rectangular extents, which in turn are intervals along appropriate dimensions of the routing space. For example, subscription region S_1 consists of one extent with two intervals: {[0.1,0.7), [0.1, 0.5)}. Intervals, and consequently interest and description expressions, are easy to capture in our model with the relations "≥" and "<", and identity function $F(x) = x$. For instance, an interest expression for subscription region S_1 may be expressed as $(X_1 \geq 0.1) \cap (X_1 < 0.7) \cap (X_2 \geq 0.1) \cap (X_2 < 0.5)$.

2.4 Approaches

Two well-known approaches to realizing DDM are to form groups based on (1) grids [4, 5, 10] and (2) update regions [1]. As will be described next, the grid-based approach provides a simple means to match update and subscription regions, but tends to utilize a large number of multicast groups, and can result in duplicate or extra messages that must be filtered at the receiver. The update region approach avoids these drawbacks, but at the cost of greater complexity (and runtime overhead) to match update and subscription regions. Each of these is described in turn.

2.4.1 Region-Based Groups

In the regions based approach, a multicast group is defined for each update region [1]. Updates are simply sent to the group associated with the update region. A federate subscribes to the group if one or more of its subscription regions overlap with the update region.

When a subscription region changes, the new subscription region must be matched against all other update regions in order to determine those that overlap with the new subscription region. The federate must then join the groups with overlapping update regions. Similarly, when an update region changes, the new update region must be matched against all subscription regions to determine the new composition of the update region's group. This requires examining all subscription/update regions in use by the federation. Thus it does not scale well as the number of regions becomes large. On the positive side, duplicate or extra messages cannot occur with this approach.

2.4.2 Grid-Based Groups

In the grid-based approach the routing space is partitioned into non-overlapping grid cells, and a multicast group is defined for each cell [4, 5, 10]. A federate subscribes to the group associated with each cell that partially or fully overlaps with its subscription regions. An update operation is realized by sending an update message to the groups corresponding to the cells that partially or fully overlap with the associated update region.

A federate may have multiple subscription regions overlapping a specific grid cell. To avoid multiple subscriptions to the group, each grid cell can maintain a subscription count array with an entry for each federate that indicates the number of subscription regions for that federate that overlap this cell. The federate leaves the group if this count becomes zero during a subscription region change. Similarly, the federate will join the group if its count becomes non-zero.

The grid-based approach eliminates the need to explicitly match update and subscription regions. While grid partitioning eliminates the matching overhead, a large number of groups are needed if a fine grid structure is defined. A coarse grid leads to imprecise filtering, negating some of the benefits of DDM. In addition, the grid scheme has other shortcomings:

- *Duplicate messages* may occur. For example, if a subscription and update region both overlap with the same two cells, two identical copies of the message will be sent to the subscribing federate over different multicast groups. These must be filtered at the receiver, incurring additional overhead.
- *Extra messages* may occur. This is a direct result of discretizing the routing space into grid cells. Subscription and update regions may overlap with the same grid cell, but may not overlap with each other. In this case, a message will be sent to the subscribing federate, even though its subscription region does not overlap with the update region. These unwanted messages will also have to be filtered at the destination.

There is a tradeoff between the number of duplicate and extra messages as the grid cell size changes. Smaller grid cells will generally result in fewer extra messages, but more duplicates, and vice versa. Any number of groups other than the one corresponding to an optimal grid cell size will produce more messages. [19] compares the cost of cell-based and entity based grouping strategies, but does not propose a solution to calculate the cell size. INRIA [20], on the other hand does propose a solution to dynamically calculate an optimal cell size for real-time simulations - large-scale virtual environments, based on the density of participants and their velocity.

Finally, in [8] multicast groups are defined for multiple update regions. Update regions are selected for inclusion in a group according to certain criteria up to the point where total message latency does not exceed maximum tolerable latency t_{max}. A chosen update region is included in the current group by having the update region's recipients (i.e. subscribing federates) added to the group. This process stops when we run out of multicast groups or cover all update regions, whichever occurs first. For the remaining update regions that were not covered, we may have to use point-to-point communication. This approach is well suited for the real-time federations, since groups are being formed in a way not to exceed the total allowable latency for message delivery. Duplicate messages cannot happen, but extra messages may occur due to the bundling of multiple update regions' recipients.

2.4.3 Implications to Time Managed DDM

The region based and grid based approaches presented here can be used in an unsynchronized DDM system. This thesis builds on these DDM approaches by adding the synchronization. Goal is to process all simulation events including the DDM events (e.g. region changes) in time stamp order at each logical process. Hence the DDM can be used for analytical simulations where causal relationships must be preserved between all simulation events including the DDM events that change the connectivity between LPs during the simulation execution. Our time managed DDM approach also preserves the repeatability of analytical simulations. This allows DDM to be used in analytical simulations that are usually used to analyze complex systems, where every simulation run must not only preserve before-and-after causal relationships but also has to produce the same final simulation state and the same outputs for the same input parameters.

Analytical simulations which use DDM typically run in an as-fast-as-possible manner. Thus, we need to keep the time-managed DDM overheads to a minimum. We also need some way to include different versions over logical time of update and subscription regions as well as the constructs and data structures that are used to perform matching, such as the grid cells described previously.

Our time managed DDM approach addresses the issue of keeping different versions of data structures with the space time memory and the way that the multicast groups are formed. Space time memory is a two-dimensional memory system used for DDM computations where we need to keep time evolving data in a one-dimensional computer memory (characterized by a common spatial memory addressing found in most computers today). Additional time dimension is introduced so that both spatial and time coordinates are used for addressing in the space time memory. Finally, we want to utilize network multicast capability for as many multicast groups found by the DDM mapping as possible. Unsynchronized DDM mapping problem was described earlier in section 2.1, while the synchronized DDM mapping problem will be presented at the beginning of the next chapter.

We have seen the tradeoffs of each of the approaches described for unsynchronized DDM. Enhancing these approaches for time managed DDM may not only exaggerate the already discussed overheads, but can also produce completely new overheads. Hence, the tradeoffs of such approaches need to be reevaluated. Our hybrid approach to matching of update and subscription regions is designed to overcome the inefficiencies of the pure region and grid based matching skims. This approach will be discussed in detail in Chapter 3.

CHAPTER 3

TIME MANAGED DATA DISTRIBUTION MANAGEMENT

This thesis is concerned with synchronizing DDM with other simulation's events. In the common discrete event simulations the connectivity between logical processes is known apriori, since it is realized inside the simulation model. In contrast, the data distribution management changes this assumption. The connectivity between logical processes is realized outside the simulation model since LPs can declare interest changes (e.g. update and subscription region changes) at different instances in logical time.

We start this chapter by discussing the problem and presenting it in a formal way by extending the model from the previous chapter to include logical time. Besides duplicate and extra messages described previously, section 3.2 talks about additional issues that need addressing. In particular, we identify the issue of missed messages, extra messages due to the synchronization, and messages in the LP's past.

In section 3.3, our comprehensive approach to synchronized DDM is presented. The optimistic approach to deal with the problems of unsynchronized DDM is presented in section 3.3.1. This is followed in section 3.3.2 by the approach to reduce matching overheads. Besides addressing the issues identified, we discuss why it compares well to other approaches. Although optimistic in nature, our approach does not suffer from inefficiencies that are common to other optimistic techniques, since it is specifically tailored to the semantics of the DDM operations.

Finally, details of the algorithm and design decisions can be found in sections 3.4 and 3.5. The algorithm section presents our two-layered DDM architecture, describing the functionality and each of the layer's operations in detail. This is followed by the pseudo-code of the algorithm and the proof of its correctness. We then summarize the design decisions and discuss the design rationale and tradeoffs for dealing with missed and messages in the past, as well as how to represent distribution lists and how to do matching of update and subscriptions regions.

3.1 Problem Description

The distinction between non-time managed services such as those employed in real-time training simulations and time managed services concerns when the service takes effect, and the order in which the service takes effect in relation to other services invoked during the federation execution. Time managed services employ a logical time abstraction giving one explicit control over the order in which these services are observed by the participating federates. On the other hand, non-time managed services are based on wallclock time which does not provide such ordering guarantees. This may be acceptable to some simulations, such as real-time training simulations where different orders of events are not perceptible to human participants.

One reason for using time-managed services is to avoid non-determinism in the federation execution, which in turn can create non-repeatable results. Directly applying training simulation DDM mechanisms based on wallclock time semantics to logical time simulations will lead to errors, as we will see in the next section.

The definition of time managed DDM is based on the semantics of DDM operations in the context of a sequential execution. A snapshot of the state, i.e., the state of all the LPs in the parallel/distributed execution at a certain logical time will be exactly the same as a snapshot of the state of a sequential simulation for that logical time.

As explained in the introductory chapter, the sequential simulation execution is characterized by the execution of events in time stamp order. At each step, the event with the smallest time stamp is taken from the list of pending events and processed. Processing an event may result in modifications of the state of the system (i.e. state of an appropriate LP) and may generate events for this or other parts of the system (i.e. this LP or other LPs). These events are enqueued in the list of pending events according to their time stamps, leaving the list of events sorted at the end of each simulation step. DDM, in essence, extends this execution style by labeling each event with the list of LPs that have to process it.

3.1.1 Formal Problem Statement

Now we extend the definitions from Chapter 2 to include logical time. Recall that the language consists of all possible expressions to specify a portion of the name space. An expression is a single test or a conjunction of tests of an arbitrarily complex function with some or all of its arguments in the name space for any possible logical time:

Expression $e(t) := (\rho_1(F_1(v), v_1) \cap \rho_2(F_2(v), v_2) \cap \ldots \cap \rho_K(F_K(v), v_K)) (t)$

That is, it is a single relation or an intersection of relations of type $\rho(F(v), v_T)$ for a given test point in the name space $v_T \in V$ and logical time t. Note that each of the relations, functions and test points may differ for different logical times. Hence, expression can evaluate to different sets of points in the name space at different logical times, where each tuple satisfies all of the binary relations:

Expression evaluation $V(e)(t) = \{ v \mid v \in V$ and $(\rho_1(F_1(v), v_1)$ and $\rho_2(F_2(v), v_2)$ and… and $\rho_K(F_K(v), v_K)) (t) \}$

There are situations when we want a relation to evaluate to TRUE or FALSE independent of the arguments. When we need a relation ρ_I to evaluate to TRUE for a particular logical time, it can be set to the "always TRUE" relation, that is no matter what the function F_I and test point v_I are, relation ρ_I is always TRUE for any point in the name space $v \in V$. Similarly, when we need a relation ρ_I to evaluate to FALSE for a particular logical time, it can be set to the "always FALSE" relation.

This allows us to describe expressions more concisely, that is as being active for the entire time line of the simulation, instead of having two or more expressions representing the evolution of a single expression. For example an interest expression in the routing spaces may be $(X < 100) \cap (Y < 90)$ for times [0, 10), then change to (X "always

FALSE” 100) $\cap$ (Y < 90) for times [10, 20) and finally (X < 200) $\cap$ (Y “always TRUE” 90) for times [20, 30). Two relations are in use. Initially, two relations specify a rectangular portion of the routing space. The first relation changes to “always FALSE” in the [10, 20) time interval, practically disabling the expression for this interval (e.g., a radar has stopped working for this time interval). The second relation is set to “always TRUE” in the [20, 30) time interval, when the expression specifies a rectangular portion of the name space with any possible value for the Y coordinate.

Definitions

- Let $\boldsymbol{P}$ represent the set of all processes in a distributed application: $\boldsymbol{P} = \{ P_1, P_2, \ldots, P_{Np}\}$ where Np is the total number of processes
- Let $\boldsymbol{E}$ represent a set of all expressions in a distributed application: $\boldsymbol{E} = \{ e_1(t), e_2(t), \ldots, e_{Ne}(t)\}$ where Ne is the total number of expressions
- Let $\boldsymbol{D}$ be a set of all description expressions such that $\boldsymbol{D} \subseteq \boldsymbol{E}$
- Let $\boldsymbol{I}$ be a set of all interest expressions such that $\boldsymbol{I} \subseteq \boldsymbol{E}$
- Let $\boldsymbol{p}$ be a mapping $\boldsymbol{p}$: $\boldsymbol{E} \rightarrow \boldsymbol{P}$ designating a process that produced an expression. For example, if process P_j produced expression e_i we can write $\boldsymbol{p}(e_i) = P_j$
- Let $\boldsymbol{r(t)}$ be a mapping $\boldsymbol{r(t)}$: $\boldsymbol{D} \rightarrow \boldsymbol{P}$ designating all consumer processes (i.e. recipients) whose interest expressions overlap a description expression. That is, for any logical time T during the execution and description expression $d(t) \in \boldsymbol{D}$, $P \in \boldsymbol{r}(d(T))$ if and only if $\exists\, i(t) \in \boldsymbol{I}$ such that $\boldsymbol{p}(i(t)) = P$ and $V(d(T)) \cap V(i(T)) \neq \emptyset$

- Let a multicast group $\boldsymbol{G_i(t)}$ be a subset of all processes, that is $\boldsymbol{G_i(T)} \subseteq \boldsymbol{P}$ for any logical time T during the execution. Three operations can be performed on a multicast group:
    1. The ***Mcast_join_group(G_i, P_j, T)*** operation adds process $\boldsymbol{P_j} \in \boldsymbol{P}$ to the multicast group $\boldsymbol{G_i}$ at logical time $\boldsymbol{T}$.
    2. The ***Mcast_leave_group (G_i, P_j, T)*** operation removes process $\boldsymbol{P_j} \in \boldsymbol{P}$ from the multicast group $\boldsymbol{G_i}$ at logical time $\boldsymbol{T}$.
    3. The ***Mcast_message (G_i, msg_content, T)*** operation sends a message with content ***msg_content*** to all processes in $\boldsymbol{G_i}$ at logical time $\boldsymbol{T}$. This is also referred to as multicasting a message to the group. ***msg_content*** contains all the necessary information about the message content, e.g., the memory location where the message content begins and the length of the message.

DDM problem statement

For a given number of multicast groups ***Ng*** and ***P***, ***E***, ***D***, ***I***, ***p*** and ***r*** find:

1. A set of multicast groups $\boldsymbol{G} = \{ G_1(t), G_2(t), \ldots , G_{\boldsymbol{Ng}}(t)\}$ where $G_i(T) \subseteq \boldsymbol{P}$ for any logical time T during the execution

    and

2. A mapping $\boldsymbol{m(t)}$: $\boldsymbol{D} \rightarrow \boldsymbol{G}$ such that for any logical time T during the execution and $\forall d(t) \in \boldsymbol{D},\ \boldsymbol{r}(d(T)) \subseteq \bigcup_{\forall G_i(T) \in \boldsymbol{m}(d(T))} G_i(T)$

The previous chapter discussed how DDM determines what data to route between information producers and information consumers. The problem statement says that for each message that is produced, DDM must determine the set of consumers interested in receiving that message. Note that this does not preclude a consumer from receiving additional messages that it had not requested to receive, but if it does, it should discard them.

In simulation terms, messages are events. So DDM in essence, labels each event with the list of LPs that must process it. This labeling of events can also be viewed as a special type of event, or DDM events. These new events are generated by our DDM approach, as will be seen later in this chapter.

3.2 Issues in Time Managed DDM

Chapter 2 introduced DDM issues such as duplicate and extra messages due to the mapping of description expressions to multicast groups, and control messages exchanged during group modifications. Since duplicate and extra messages can easily be filtered when they are received by LPs, all three types of messages can be seen as DDM overhead. In addition to these overheads, synchronized DDM may raise other issues. Synchronization of DDM and non-DDM events is necessary, otherwise the following types of errors may occur: (1) missed messages, or (2) extra messages due to the synchronization, and (3) messages in the LP's past. We will examine these issues now.

Missed messages in non-synchronized DDM. Let the mapping *m(t)* at logical time T indicate that logical process P_j is the recipient of messages from logical process P_i that are labeled with description expression *d(t)*. Then, all such messages at time T should be sent to P_j. However, the mapping *m(t)* may not yet be completed when a message is sent due to “late” description and interest expressions that are generated with a logical time less than T. Such a message may “miss” LP P_j. “Late” expressions can occur since DDM events which are transmitted as messages between LPs can consequently be delayed due to network latencies. This can also happen when a message at time T is sent before the relevant expression at time T was even issued.

To illustrate the situation where missed messages occur, consider two tanks T1 (red tank) and T2 (blue tank) as depicted in Figure 2. If T1’s sensors indicate awareness of an enemy tank T2 in a certain time interval, all events generated by tank T2 with time stamp in that interval must be received by tank T1. It may happen that tank T2 generated events in this interval before (in wallclock time) T1 had specified interest in receiving these events. This will result in missed messages that should have been received by tank T2. Proper synchronization must ensure these previously generated events are sent to T1.

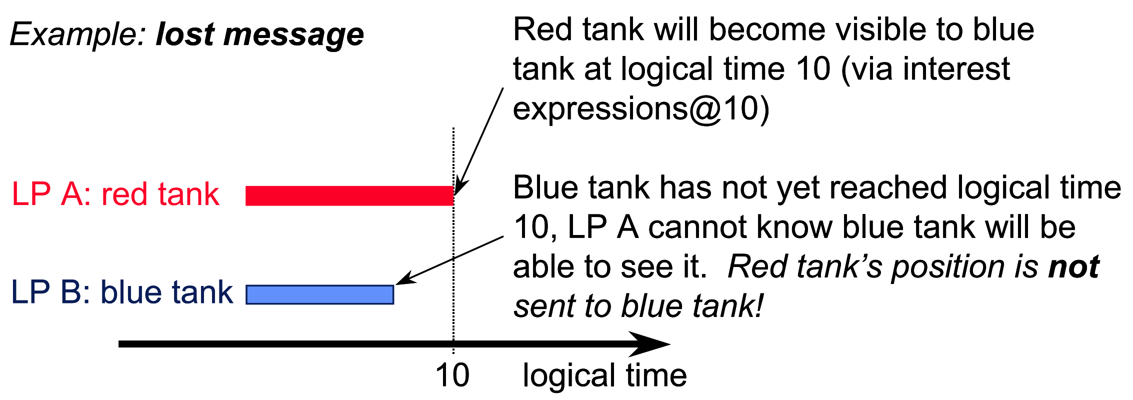

Example: **lost message**
Red tank will become visible to blue tank at logical time 10 (via interest expressions@10)
LP A: red tank
Blue tank has not yet reached logical time 10, LP A cannot know blue tank will be able to see it. *Red tank's position is **not** sent to blue tank!*
LP B: blue tank
10
logical time

Figure 2 Missed messages example.

Extra messages in non-synchronized DDM. Let the mapping *m(t)* at logical time T indicate that logical process P_j is not the recipient of messages from logical process P_i that are labeled with description expression *d(t)*. Then, all such messages at time T should not be sent to P_j. However, the mapping *m(t)* may not yet be completed when a message is sent due to "late" description and interest expressions, as described above, that affect logical time T. Such a message may actually be sent to LP P_j. Extra messages present only a performance burden since they can be filtered at the destination LPs. Otherwise, if we process all events regardless of whether they should have been received or not, simulation state may become corrupt, and repeatability may be lost.

A scenario similar to the above one may be constructed to illustrate the extra message problem. If T1's sensors cannot detect an enemy tank T2 in a certain time interval, all events generated by tank T2 with time stamp in that interval should not be received by tank T1. It may happen that tank T2 generated events in this interval before (in wallclock time) T1 had moved out "late" from its previous position when it had been specifying interest in receiving these events. This will result in extra messages that should have not been received by tank T2. Proper synchronization must ensure that such messages are never delivered to tank T2.

Messages in the LP's logical time past in non-synchronized DDM. Computations that handle time advances in parallel and distributed simulation (also known as LBTS computation) determines the current simulation time by taking into account all connections between LPs. A common assumption in approaches to LBTS computation is that these connections are static, i.e., they do not change over logical time. DDM, however, violates this assumption by adding and ceasing connections between LPs at different logical times as description and interest expression change. Hence, the problem arises when a DDM event adds a new connection LP_{src} - LP_{dest} which was not taken into account by the latest LBTS computation. Now, LP_{src} may send a message to LP_{dest} in its logical time past, which is unacceptable.

This issue is concerned with ensuring proper synchronization of changes in the connection topology among LPs. For example, suppose in the previous example T2 is at logical time 6. Suppose T1 advances beyond logical time 6 because there is no connection from T2 to T1. Suppose T2 now establishes a new connection to T1 at logical time 6, and sends a message with time stamp 6, in T1's logical time past. This is depicted in Figure 3. Situations such as this must be prevented.

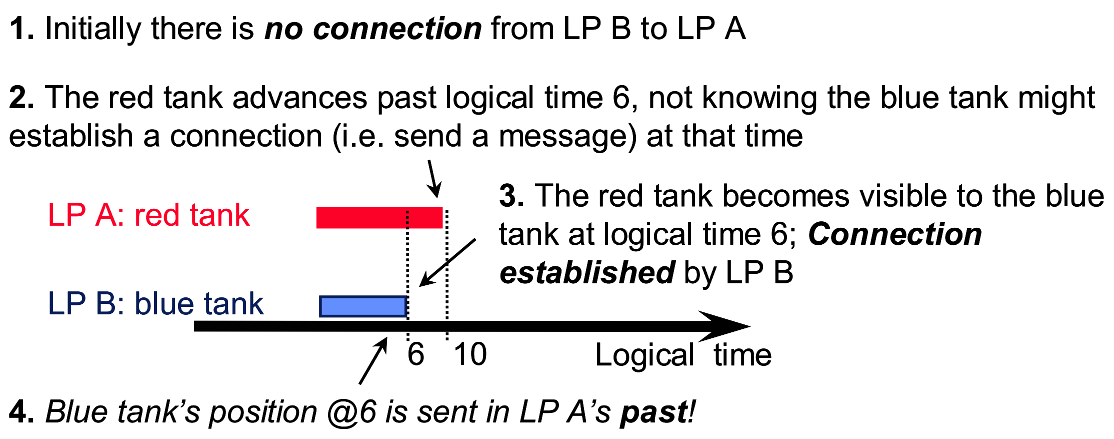

1. Initially there is no connection from LP B to LP A
2. The red tank advances past logical time 6, not knowing the blue tank might establish a connection (i.e. send a message) at that time
LP A: red tank
3. The red tank becomes visible to the blue tank at logical time 6; Connection established by LP B
LP B: blue tank
6
10
Logical time
4. Blue tank's position @6 is sent in LP A's past!

Figure 3 Example of messages in the LP's past.

In summary we may conclude that without time managed DDM, missed, extra and past messages may occur in addition to the duplicate, extra and control messages described in the previous chapter. All these situations must be avoided. Otherwise, parallel and distributed simulation may not produce the same results as the corresponding sequential execution. Furthermore, executions may not be repeatable.

3.3 Synchronized DDM Approach

We will now describe our approach to time managed DDM. As we have discussed so far, when using DDM for analytical distributed simulations an LP's interests change dynamically (e.g. position of a sensor changes), and they are interleaved with generating events, not necessarily in time stamp order. Since these DDM changes may not arrive in time stamp order synchronization is required, so that the end result is that each LP processes all its events in time stamp order.

Our approach for synchronizing simulation and DDM events addresses the issues described in section 3.2, that is, how to deal with the missed messages, extra messages and messages in the past. This approach is optimistic in nature since it allows events to be processed out of time stamp order, and an error detection and recovery procedure is used rather than strictly avoiding errors. Furthermore, it is tailored to the semantics of the DDM services, thereby avoiding the use of general rollback mechanisms, which are more complex and require large runtime overheads. The optimistic approach will avoid non-determinism for analytical simulations which use DDM, and hence will create repeatable simulations and the results of such simulations.

Furthermore, our approach also addresses the issue of how to reduce DDM overheads. Lowering DDM overheads for both time managed and non-time managed distributed systems is important to improve the scalability for any such distributed systems. In particular, we address the problem of matching update and subscription regions with the hybrid approach that reduces DDM overheads. In this approach we use counters in the grid partitioned routing space to achieve fast transformation of multiple object interests into the connectivity.

The following two sections describe our comprehensive approach to time managed DDM. Section 3.3.1 presents the optimistic approach for dealing with issues in unsynchronized DDM. Section 3.3.2 presents the hybrid approach for reducing DDM overheads.

3.3.1 An Optimistic Algorithm to Time Managed DDM

A solution to the missed messages problem is to provide a message log to record messages that otherwise would be missed. Messages are logged as they are sent. When a new DDM event indicates that a previously generated message should have been sent to an LP, but in fact was not, the message is retrieved from the log and sent.

A solution to the extra messages problem is the same as that which was discussed earlier. Extra messages can be avoided by performing extra filtering by the LPs receiving the updates.

Finally, the solution to the problem of messages in the LP's past is to modify the LBTS computation which is responsible for each of the LPs' time advances. Recall that the LBTS computation determines the current simulation time to which an LP can advance (i.e. events with time stamps up to that time are safe to process), by taking into account all connections between LPs. However, some of those events may be "late" DDM events, which have been delayed in the network as transient messages. The problem is that these DDM events may generate new messages from the log with time stamps less than the new simulation time determined by the LBTS computation. It is straightforward to correct this problem by having the LBTS computation take into account any "late" DDM events with time stamps less than the computed LBTS value. In particular, each "late" DDM event with time stamp less than the new LBTS value must be processed immediately, out of time stamp order, and all generated update messages must be accounted as transient messages in the extended LBTS computation. Note that the LBTS computation should not impose any additional overheads if there are no “late” DDM events with time stamps less than the computed LBTS value.

To illustrate the log-based approach, consider the scenario shown in Figure 4. The red tank modeled by LP_A issues an update position event at logical time 10. The blue tank modeled by another LP_A is not subscribed to receive any messages from LP_B at this time. Later in the execution, suppose the blue tank changes its position at logical time 9 and hence becomes subscribed to events from LP_A at that logical time. The previously logged event from LP_A is retrieved from the log, and sent to LP_B.

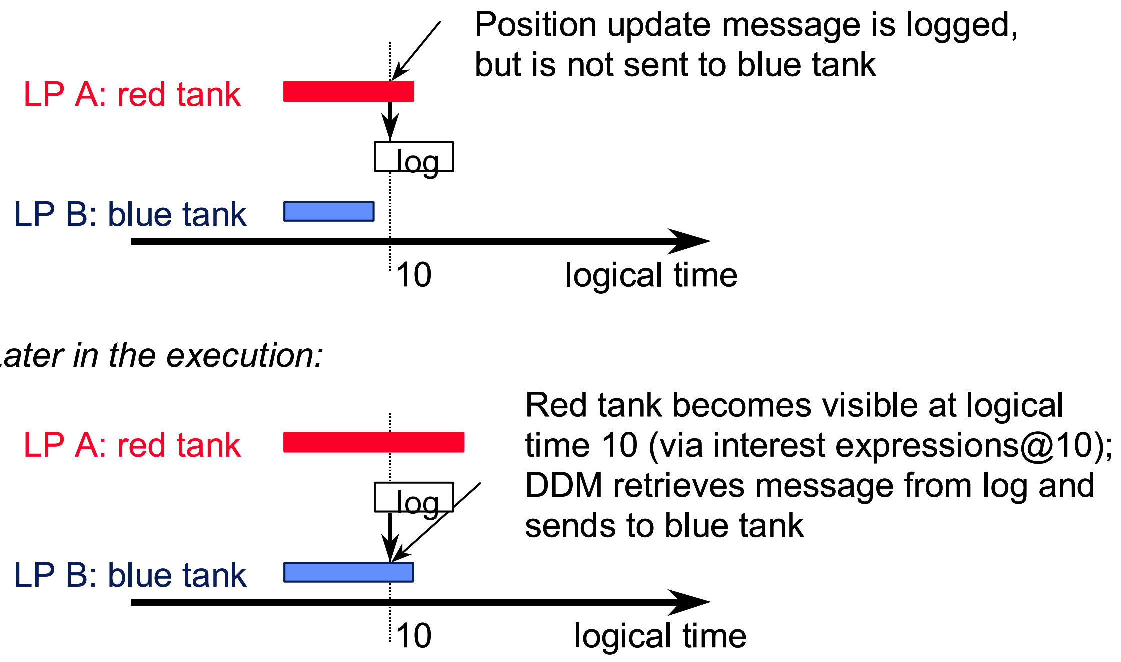

Position update message is logged, but is not sent to blue tank
LP A: red tank
log
LP B: blue tank
10
logical time
Later in the execution:
Red tank becomes visible at logical time 10 (via interest expressions@10); DDM retrieves message from log and sends to blue tank
LP A: red tank
log
LP B: blue tank
10
logical time

Figure 4 Message log to prevent missed messages.

This approach is optimistic in that an error detection and recovery procedure is used rather than strictly avoiding errors. By contrast, a conservative execution would processes events, including DDM events, in time stamp order at each LP, and ensure that there will not be any messages in the LP's past, nor any missed messages. Here, DDM events may be processed out of time stamp order. In fact these events are processed as early as possible. Errors such as missed messages may occur, and when detected, the log is used to recover.

The advantage of this optimistic approach is it allows greater concurrent execution of events, and avoids the inconvenience of requiring each LP to always invoke services that affect the connection database ahead of time. In other words, zero lookahead interactions and concurrent execution are both allowed. A second advantage is that this particular optimistic approach to synchronizing the DDM services is tailored to the semantics of the DDM services, thereby avoiding the use of general rollback mechanisms, which are more complex and potentially require large runtime overheads. The main drawback with optimistic methods is the additional complexity it introduces in the simulation infrastructure. In the example that was just described the infrastructure must maintain message logs, and provide mechanisms to reclaim storage used by the log.

Related to this optimistic approach is the one described in [29]. The "optimistic publications" mechanism is proposed that allows publishers to go ahead of subscribers more freely. This is similar to our approach since publishers keep history of sent messages. However, rollbacks are utilized to handle delayed subscriptions at the publishers. Furthermore, subscribers wait for "catch-up" messages and handshake with publisher during subscription events, while in our approach there is no wait on subscription events, until possibly during the LBTS computation. Even then, the LBTS computation does not impose any additional overheads such as handshakes with publishers, unless there are “late” DDM events with time stamps less than the computed LBTS value.

Another related approach is described in [27]. It differs from our approach since it is based on an optimistic simulation, and thus has the complexity and performance burden inherent to the optimistic style of execution.

In conclusion, our approach to time managed DDM consists of two parts. One part is the optimistic approach to synchronization between DDM and non-DDM events. The second part is the hybrid approach to DDM, which deals with the issue of reducing computational (i.e. matching) overheads necessary for DDM to determine subscribers for each update region. This approach is applicable to both time managed and non-time managed DDM systems, and will be described in the following section.

3.3.2 Hybrid Approach to Reduce DDM Overheads: Region-Based Groups with Grids

The time managed DDM algorithm described here uses a hybrid approach to DDM. This hybrid approach uses a variation on the region-based approach described in Chapter 2 that uses grid cells to reduce the matching overhead. A multicast group is defined for each update region, eliminating the duplicate and extra message problem of the grid scheme (for non-time managed DDM). However, grid partitioning is used to match update and subscription regions, improving the scalability of the pure update-region based approach.

Grids can be used to improve the efficiency of region changes. Logically, when a subscription region changes, one need only consider those update regions overlapping the grid cells covering the old and new subscription regions to determine the new composition of multicast groups. Similarly, when an update region changes, one need only consider those subscription regions that overlap the grid cells of the old/new update region to determine the new composition of the group.

We use a variation on this approach to manage group membership. Recall the pure grid-based approach used subscription counts to track the number of times a logical process is subscribed to a grid cell. The hybrid approach uses a similar concept, but for update regions, to trigger group join and leave requests. Specifically, a *subscription strength* array is defined for each update region, with one entry per LP. The entry for a LP indicates the "strength" of that LP's subscription to the update region (group). One unit of strength corresponds to one subscription region for the LP overlapping with the update region in exactly one grid cell. The strength of a subscription region is the number of grid cells in which the subscription region overlaps with the update region. The total strength of the LP's subscription to an update region is the sum of the strengths of each of the LP's subscription regions. For example, if the LP has two subscription regions, and one overlaps the update region in one cell, and the second overlaps it in two cells, the strength of the LP's subscription to the update region is three. The LP remains joined to the update region's multicast group so long as it has a subscription strength of at least one. The DDM software maintains the strength arrays as regions come and go and are modified. It issues a join request if the LP's subscription strength becomes non-zero, and issues a leave request if the strength becomes zero.

This approach is easily extended to consider classes and attributes, as required in the HLA DDM services. It was first described in [6], which coincided with a similar idea described in [7]. Our hybrid approach differs in that it does not require a centralized coordinator for matching update and subscription regions. Another difference is that our approach is more general since it allows multiple subscription regions to be treated as a single interest expression per LP, as oppose to having each subscription region as an interest expression. On the other hand, it is less precise since there may be extra messages generated due to the granularity of grid cells. This may happen when regions do not overlap grid cells fully, but only partially.

3.4 Algorithm

As discussed in Chapter 2, the fundamental concept underlying interest expressions is the *routing space.* A routing space is a normalized multidimensional coordinate system in which LPs indicate interest in receiving or providing updates via *subscription* and *update regions*, respectively. Regions are rectangular (in N dimensions) and are specified by indicating *extents,* with one extent for each dimension. Each extent indicates the portion of that dimension covered by the region. A federate (i.e. an LP) may issue **Modify_Region** for changing region extents and **Update_Attribute_Value** to send messages (see [30] for complete reference to HLA services). When an attribute is updated, a message is sent only to those federates whose subscription region overlaps with the update region.

Here, we describe one approach to implementing the time managed DDM algorithm based on partitioning the routing space into fixed non-overlapping cells and creating a distribution list for each update region by combining the information from cells which are overlapped by that region. The core of this approach was described in sections 3.3.1 and 3.3.2. Here, we will assume identical cells to simplify the presentation.

3.4.1 DDM Architecture

It is convenient to view the DDM system as logically being composed of two layers (see Figure 5). The upper layer provides the interface to the federate for specifying its interest and description expressions (portions of the routing space). This interest management layer receives and processes these expressions and generates for the lower layer *Add* and *Delete* operations to change the database indicating which federates receive attribute updates and interactions. At this lower layer, distribution list software performs changes to the database and ensures that these changes are properly synchronized with attribute updates and interactions so that each federate receives all of the messages it is supposed to receive, and no others.

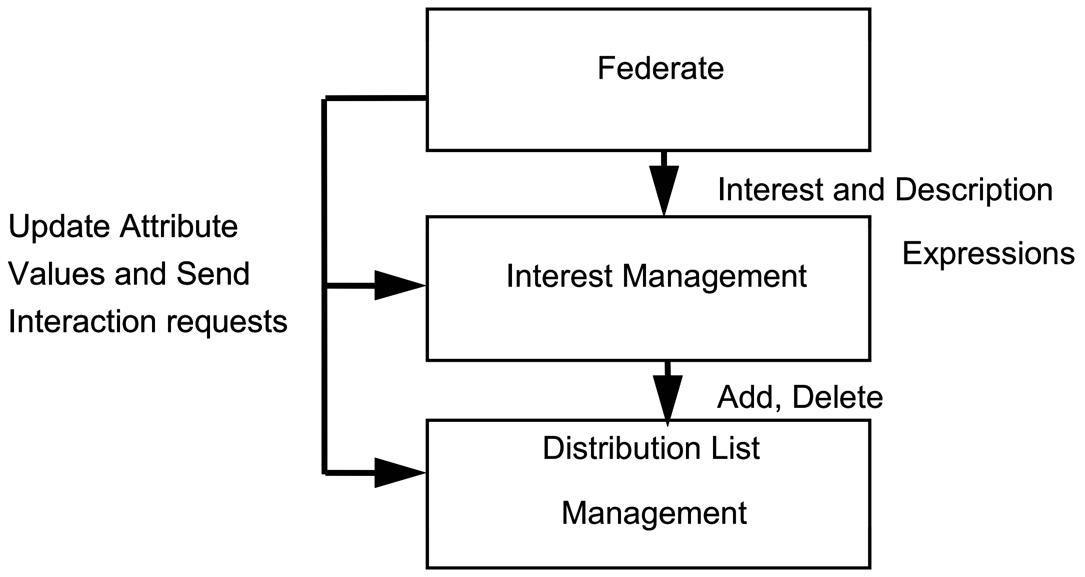

Federate
Interest and Description Expressions
Update Attribute Values and Send Interaction requests
Interest Management
Add, Delete
Distribution List Management

Figure 5 DDM Organization.

3.4.2 Interest Management Layer

The current HLA Interface Specification [30] does not include a time stamp parameter for services that modify subscription and update regions. It is clear that such a specification is necessary for logical time federations to indicate when the changes should take effect, so here, a time stamp parameter has been added for this purpose.

The interest management layer provides the following data distribution services to the federate:

- **region_handle* Create_Region (space_handle theSpace, ULong number of extents):** A region is created by this service. The space_handle parameter must be one of the already defined routing spaces, and the other parameter is the number of extents (i.e. rectangular portions of the routing space). The region handle is passed to the federate when the region is created.

- **void Delete_Region (region_handle H, time_stamp T):** A region H is deleted by this service at logical time T.
- **void AssociateRegionForUpdates (region_handle H, object_handle theObject, AttributeHandleSet theAttributes, time_stamp T):** This service associates the region H to be used for updates with instance attributes theAttributes of a specific object instance theObject at the specified logical time T. In essence, this service makes the update region H active (i.e. to be considered for data distribution) at logical time T.
- **void UnassociateRegionForUpdates (region_handle H, object_handle theObject, time_stamp T):** Removes the association between the region H and all instance attributes from object theObject associated with that region at time T.
- **void SubscribeObjectClassAttributesWithRegion (ObjectClassHandle theClass, region_handle H, AttributeHandleSet attributeList, time_stamp T):** Makes an association between the object class theClass with attributes theAttributes and region H. Basically, this service makes the subscription region H active (i.e. to be considered for data distribution) at logical time T.
- **void Unsubscribe ObjectClassAttributes WithRegion (ObjectClassHandle theClass, region_handle H, time_stamp T):** Removes the association between the object class theClass and the region H at time T.

- **void Modify_Region (region_handle H, time_stamp T):** Informs the RTI that the region H has been changed to a new set of extents pointed to by the region_handle H as of logical time T.
- **void Update_Attribute_Values (object_handle theObject, AttributeHandle ValuePairSet theAttributes, time_stamp T):** Informs the RTI of a new attribute values contained in the theAttributes at logical time T.

For the sake of completeness, federates may also issue declaration management operations (*Publish* and *Subscribe*). These operations are processed in the interest management layer, but are not discussed further.

Now, we describe in more detail our hybrid approach to implementing the interest management layer based on partitioning the routing space into non-overlapping cells. In general, the cells need not be the same size, but we will assume identical cells here to simplify the presentation.

As discussed in section 3.3.2, grid partitioning is used to match update and subscription regions, and a multicast group is defined for each update region. A federate may be subscribed to a cell multiple times, e.g., if more than one entity within the federate has indicated subscriptions to regions overlapping the same cell. For this reason, counters are used to indicate the strength of a federate's subscription to the cell.

To explain how the interest management layer works, we first define two types of counters:

- ***Def C1.*** *(Cell counter) Grid cell's subscription strength per federate, denoted $C_{sbsc}[F]$: strength@t.* This counter keeps track of how many subscription regions of a federate overlap a cell. C_{sbsc} refers to a grid cell, F is a federate subscriber and t is a logical time.
- ***Def C2.*** *(Region counter) Cumulative subscription strength per update region, denoted $U_{cumulative}[UR][F]$: strength@t.* This counter is what we were referring to as subscription strength for an update region in our hybrid approach. In other words, the cumulative subscription strength at a logical time t is a sum of subscription strengths for this region of the grid cells overlapping an update region. *UR* is an update region, F is a federate subscriber and t is a logical time. In other words:

$$U_{cumulative}[UR][F]: strength@t = \sum C_{sbsc}[F]: strength@t$$
$$\forall \text{ overlapping cells by region } UR \text{ at time } t$$

Essentially, the algorithm updates these two types of counters as regions change, and generates time-stamped Add/Delete operations for the Distribution List Layer. During this process, when $U_{cumulative}$ counter's value increases to 1, an Add operation is generated, and when it drops to 0, a Delete operation is generated.

The *ModifyRegion* service for an update region must compute a new value of the $U_{cumulative}$ counter for that region. Instead of using the above formula, optimizations to improve performance are possible. When an update region changes, it is not necessary to recompute the sum of C_{sbsc} for all cells that were covered by a previous instance of this region and remain covered by the region after the change. The same result can be obtained by adding to the $U_{cumulative}$ counter C_{sbsc} for the cells not previously covered (i.e.

new cells) and subtracting C_{sbsc} for the left cells (i.e. old cells). So, we use the following formula to improve performance:

$$U_{cumulative}[UR][F]: strength@t = \sum_{\forall \text{ overlapping cells at time } t} C_{sbsc}[F]: strength@t$$

$$= \sum_{\forall \text{ overlapping cells from previous instance of } UR} C_{sbsc}[F]: strength@t \quad \text{// speed up with incremental update}$$

$$+ \sum_{\forall \text{ new cells}} C_{sbsc}[F]: strength@t \; - \sum_{\forall \text{ old cells}} C_{sbsc}[F]: strength@t$$

The *ModifyRegion* service for a subscription region has to update C_{sbsc} counters only for the affected cells. That is, counters for newly overlapped cells (i.e. new cells) need to be incremented by 1, while counters for left cells (i.e. old cells) need to be decremented by 1. Then, we need to compute new values of the $U_{cumulative}$ counters for all update regions covering any of the new or old cells. Instead of using the original formula, an optimization to improve performance is again possible. When an update region changes, it is not necessary to recompute the sum of C_{sbsc} for all cells that are covered by the region. The same result can be obtained by adding to the $U_{cumulative}$ counter the number of new overlapping cells and subtracting the number of old overlapping cells in the subscription region. So, we use the following formula to improve performance:

$$U_{cumulative}[UR][F]: strength@t = \sum_{\forall \text{ overlapping cells at time } t} C_{sbsc}[F]: strength@t$$

$$= U_{cumulative}[UR][F]: strength@t \quad \text{// speed up with incremental update}$$

$$+ \text{ number of new overlapping cells } - \text{ number of old overlapping cells}$$

Finally, when the *UpdateAttributeValue* service is invoked, it causes an invocation of the Update service in the distribution list layer for the corresponding update region.

3.4.3 Distribution List Management Layer

Logically, the distribution list management layer can be viewed as maintaining a collection of *distribution lists* indicating which federates should receive which attribute updates. The distribution lists collectively form the database mentioned earlier. The distribution lists may not be explicitly represented within the RTI, i.e., they could be realized by the composition of multicast groups in the underlying network. Each change to a distribution list has a logical time associated with it indicating when that change takes effect. Thus, one may conceptually view each distribution list as evolving, one change at a time, over successive logical times.

Let D(id,T) denote a distribution list (the id field specifies a particular list) corresponding to federate time T. When an update to an attribute occurs with time stamp T, the interest management layer will map this attribute to one or more distribution lists, and the RTI will send a message to each federate in D(id,T) of all selected lists. The following distribution management layer operations are needed (see Figure 5):

- **Add(F,id,T):** add federate F to the distribution list identified by id to take effect at logical time T.
- **Delete(F,id,T):** remove federate F from the distribution list identified by id as of logical time T.

- **Update(id, V, T):** send a message to each federate that belongs to the distribution list id at logical time T. V indicates the value of the message. This primitive would be invoked when the federate invokes the **Update Attributed Values** (or **Send Interaction**) service.

These operations are invoked within the RTI by the interest manager. They may be invoked when a change in regions occur that changes the set of federates that should receive messages, when an object becomes "discovered" (or removed), when a new object is instantiated (or removed), or when an attribute is updated or an interaction is sent. The semantics of these operations are defined as follows:

- Composing the operations **Add(F,id,T)** and **Delete(F,id,T)** with the same parameter values has the same effect as if neither operation were performed. In this case, the two operations are said to be *canceled.*
- **Add (F,id,T)**: let T_D be the smallest time stamped Delete operation (ignoring canceled operations) by federate F on distribution list id such that $T_D > T$. F will receive a message for every update to this distribution list with time stamp in the interval $[T, T_D)$.
- **Delete (F, id, T)**: let T_A be the smallest time stamped Add operation (ignoring canceled operations) by federate F on distribution list id such that $T_A > T$. F will *not* receive any messages for updates to this distribution list with time stamp in the interval $[T, T_A)$.

- The “effect” of an **Add(F,id,T)** or **Delete(F,id,T)** operation spans the interval [T, T_{AD}], where T_{AD} is the time stamp of the smallest Add or Delete operation such that $T_{AD} > T$. Outside this interval, the operation has no effect. Note that this interval may change (be shortened) during the simulation, as the new Add and Delete operations are issued. In other words, there is no cumulative effect of these operations on a distribution list membership of federate F for any particular logical time.

The distribution list for cell id corresponding to time T is constructed by determining which federates have subscribed via the Add operation to receive updates with time stamp T. Specifically, define the “subscription function” as follows:

S(F, id, T)

= TRUE if an uncanceled operation Add (F, id, T_A) exists such that $T_A \leq T$ and no uncanceled operation Delete (F, id, T_D) exists such that $T_A < T_D < T$.

= FALSE otherwise

D(id,T) is defined as the set of federates F_i such that S(F_i, id, T) is true.

As discussed earlier, update attribute messages are logged by the RTI so they can be sent to “late” subscribers. A log L(id) for distribution list id is defined for this purpose. This log is defined as a sequence of tuples <V_i, T_i> where V_i is the new value contained in the update message sent for an attribute update with time stamp T_i.

Example. It is convenient to view the history of Add/Delete operations to an attribute by a federate according to a diagram such as that shown in Figure 6. Consider a new Add or Delete operation with time stamp T. The status of the federate (subscribed or unsubscribed) at time T only depends on the largest time stamped uncanceled Add or Delete operation with time stamp smaller than T. For example, in Figure 6 consider a Delete operation with time stamp 20. The Add operation at time 15 indicates the federate is subscribed to the attribute at time 20, independent of what other Add or Delete operations occurred with time stamps less than 15. Similarly, the effect of the new operation at time 20 only persists until the next higher time stamped uncanceled Add or Delete operation. In Figure 6, the effect of the new operation at time stamp 20 only persists until time 25. No operation with time stamp larger than 25 is affected by this new operation.

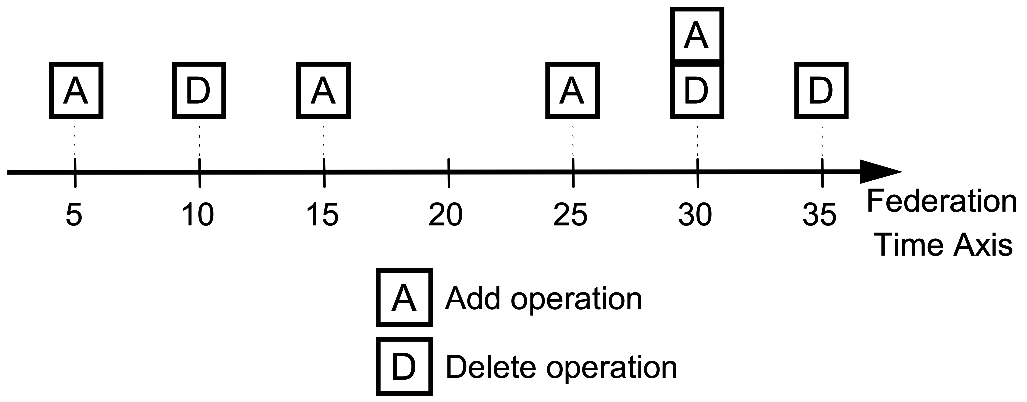


Figure 6 Snapshot of Add and Delete operations by a single federate. The federate is subscribed to receive updates in the intervals [5,10) and [15,35).

3.4.4 Basic Algorithm

Data Structures at glance.

- Cell Layer's counters:

  C_{sbsc}[F]: strength@t - grid cell's subscription strength for federate F at time t

- Region Layer's counters:

  $U_{cumulative}$[UR][F]: strength@t - cumulative subscription strength for update region UR and for federate F at time t

Without loss of generality, we keep cell and region counters for every update and subscription region change. Counters for other logical times can be deduced from existing counters as necessary. In essence, the pseudo-code below shows the underlying principles behind the algorithm in a concise way. Full description of our time-managed

DDM algorithm with all the details of how to update cell and region counters in certain logical time intervals can be found in the appendix.

Assumptions.

- Update and subscription regions always completely overlap a routing space's grid cell or not at all. There is never a grid cell overlapped partially by a region.
- Neither an update nor a subscription region change of extents may be cancelled or changed at a particular logical time where such change occurred previously. However, change is allowed to occur in between and around already issued extent changes. In other words, only one ***Modify subscription region*** or ***Modify update region*** for a region and federate may be issued for a particular logical time.

Notational abbreviations.

- For the sake of clarity, we describe region modifications for update and subscription regions separately and with slightly modified arguments. These two operations represent the *Modify_Region* service introduced previously. Furthermore, the ***subscription region SR*** or ***update region UR*** parameter of the **Modify Region** operations is in fact a region handle or a data structure that also points to, or contains, a new region's extents.
- The ***federate*** parameter ***F*** of **Modify Region** operations represents a federate which owns a particular subscription or update region. Federate F is responsible for issuing **Modify Region** operations for this region.

Algorithm.

```
Modify subscription region( subscription region SR, federate F, logical time T){
1. Phase 1: Update Cell Layer counter
    • Determine all new/old cells C for SR   // from SR(T) and previous instance in time of SR
    • Determine $T_{end}$; $T_{start}= T$
    • $\forall$ new cells:
        ++ $C_{sbsc}[F]$: strength@t ; $t \in [T_{start}, T_{end})$
    • $\forall$ old cells:
        -- $C_{sbsc}[F]$: strength@t ; $t \in [T_{start}, T_{end})$

2. Phase 2: Update Region Layer counter
    • $\forall$ UR overlapping any of the new/old cells C in $t \in [T_{start}, T_{end})$:   // limits search of URs
        $U_{cumulative}[UR][F]$: strength@t = $\sum C_{sbsc}[F]$: strength@t
                                 $\forall$ overlapping cells at time t
            = $U_{cumulative}[UR][F]$: strength@t   // speed up with incremental update
            + number of new overlapping cells – number of old overlapping cells

3. Phase 3: Issue Add/Delete operations according to change in Region Layer counter
    • $\forall$ UR where $U_{cumulative}[UR][F]$: strength@t changes its value from 0 to greater than 0 in $t \in [T_{start}, T_{end})$:
        issue **Add**( F, UR, t)
    • ∀ UR where $U_{cumulative}[UR][F]$: strength@t changes its value from greater
      than 0 to 0 in $t \in [T_{start}, T_{end})$:
        issue **Delete**( F, UR, t)
}
```

Modify update region(update region UR, logical time T){

```
1. Phase 1: Update Region Layer counter
    • Determine all new/old cells C for UR
    • Determine $T_{end}$; $T_{start} = T$
    • ∀ F in $t \in [T_{start}, T_{end})$:
```

$$U_{cumulative}[UR][F]: strength@t = \sum_{\forall \text{ overlapping cells at time } t} C_{sbsc}[F]: strength@t$$

$$= \sum_{\forall \text{ overlapping cells from previous instance of } UR} C_{sbsc}[F]: strength@t \quad // \text{ speed up with incremental update}$$

$$+ \sum_{\forall \text{ new cells}} C_{sbsc}[F]: strength@t \; - \sum_{\forall \text{ old cells}} C_{sbsc}[F]: strength@t$$

```
2. Phase 2: Issue **Add/Delete** operations according to change in Region Layer counter
    • ∀ F where $U_{cumulative}[UR][F]$: strength@t changes its value from 0 to greater
      than 0 in $t \in [T_{start}, T_{end})$:
        issue **Add**( F, UR, t)
    • ∀ F where $U_{cumulative}$[UR][F]: strength@t changes its value from greater than
      0 to 0 in $t \in [T_{start}, T_{end})$:
          issue **Delete**( F, UR, t)
}
```

***Distribution List Layer (DL)** operations – Unicast realization:*

```
  DL::Update( UR, V, T){
      • send V to all federates in D(UR,T)
      • record <V,T> in L(UR)
  }
  DL::Add( F, UR, T){
      • record F has been added to D(UR) at time T
      • message = empty
      • if not S(F,UR,T) then
          • let AD be the smallest time stamped uncanceled Add or Delete operation
            for F on UR with time stamp greater than T, and let $T_2$ be the time stamp
            of AD.
          • for each tupple <V,$T_v$> in L(UR) where $T \le T_v < T_2$, add V to message
          • send message to F
  }
```

```
DL::Delete( F, UR, T){
        • record F has been deleted from D(UR) at time T
        • message = empty
        • if S(F,UR,T) then
            • let AD be the smallest time stamped Add or Delete operation for F on A
              with time stamp greater than T, and let $T_2$ be the time stamp of AD.
            • for each tupple <V,$T_v$> in L(UR) where $T \le T_v < T_2$, add a retraction of V
              to message // retracting a tuple <V,$T_v$> effectively cancels it at the
              receiving federate F, as the tupple was never issued in the first place
            • send message to F
    }
```

Distribution List Layer (DL) *operations – Multicast realization:*

```
DL::Update( UR, V, T){
        • message = empty
        • message.values = V; message.destination = all federates in D(UR,T)
        • send message to MG(UR)
        • record <V,T> in L(UR)
    }
DL::Add( F, UR, T){
        • record F has been added to D(UR) at time T
        • message = empty
    • if not S(F,UR,T) then
        • let AD be the smallest time stamped uncanceled Add or Delete operation
          for F on UR with time stamp greater than T, and let $T_2$ be the time stamp
          of AD.
        • for each tupple $<V,T_v>$ in L(UR) where $T \leq T_v < T_2$, add V to
          message.values
        • Message.destination = F
        • send message via unicast to F
        • if( $F \not\subseteq$ MG(UR))
            issue JOIN_MCAST_GROUP( MG(UR), F)
  }
```

```
DL::Delete( F, UR, T){
    • record F has been deleted from D(UR) at time T
    • message = empty
    • if S(F,UR,T) then
        • let AD be the smallest time stamped Add or Delete operation for F on A
          with time stamp greater than T, and let $T_2$ be the time stamp of AD.
        • for each tupple $<V,T_v>$ in $L(UR)$ where $T \leq T_v < T_2$, add a retraction of V
          to message.values message // retracting a tuple $<V,T_v>$ effectively
          cancels it at the receiving federate F, as the tupple was never issued in
          the first place
        • send message via unicast to F
        • if($\neg\exists\ t \in$ [Current logical time, $\infty$) such that **S(F, UR, t)**)
                issue LEAVE_MCAST_GROUP( MG(UR), F)
    }

**DL::Time advance( T){**
        • $\forall F$:
          if( ( $F \in MG(UR)$) and ($\neg\exists\ t \in [T, \infty)$ such that **S(F, UR, t)**))
                issue LEAVE_MCAST_GROUP( MG(UR), F)
    }
```

3.4.5 Correctness Proof

We will now verify the correctness of the algorithm. First we provide the definitions for interest and description expressions. This is followed by a definition of the subscription function obtained by the algorithm and the correct subscription function which is equivalent to *r(t)* defined earlier. *r(t)* maps a description expression to a set of consumer processes, designating all consumer processes whose interest expressions overlap a description expression. Theorem 1 showing the correctness for the DDM algorithm follows from its supporting lemmas. Essentially, it proves that a subscription function obtained by our algorithm is the same as the correct subscription function at the end of the simulation execution.

Def 1. (Expressions) Let ***MR*** be a set of region modifications over time. An element of MR is a *tuple < region R, extents E, logical time T*> indicating new extents E of a region R at time T.

Def 2. (Interest Expressions) Let ***MSR*** be a ***MR*** set of subscription region modifications over time. This set is equivalent to ***I***, the set of interest expressions from the general DDM model.

Furthermore, let $\boldsymbol{MSR_{final}}$ be a ***MR*** set of all subscription region modifications over time during the simulation execution.

Def 3. (Description Expressions) Let ***MUR*** be a ***MR*** set of update region modifications over time. This set is equivalent to ***D***, the set of description expressions from the general DDM model.

Furthermore, let $\boldsymbol{MUR_{final}}$ be a ***MR*** set of all update region modifications over time during the simulation execution.

Def 4. (Subscription function obtained by a DDM algorithm) Let a subscription region modification set be ***MSR*** and update region modification set be ***MUR***. After executing all ***Modify subscription region*** operations corresponding to elements in ***MSR*** set, and all ***Modify update region*** operations corresponding to elements in ***MUR*** set, the "subscription function" ***S*** is defined as:

S(federate F, update region UR, logical time T)

= TRUE if an uncanceled operation ***Add(F, UR, T_A)*** exists such that $T_A \leq T$ and no uncanceled operation ***Delete(F, UR, T_D)*** exists such that $T_A < T_D < T$.

= FALSE otherwise

Furthermore, let ***S_{final}*** denote a subscription function ***S*** after all update and subscription region modifications over time during the simulation execution, that is for ***MUR_{final}*** and ***MSR_{final}***.

Def 5. (Correct subscription function – equivalent to mapping r from the general DDM model) Let a subscription region modification set be ***MSR*** and update region modification set be ***MUR***. The "correct subscription function" ***$S_{correct}$*** is defined as:

$S_{correct}$(federate F, update region UR, logical time T)

= TRUE if $\exists$ UR $\in$ ***MUR*** and $\exists$ SR $\in$ ***MSR*** at time T such that SR is produced by federate F and UR $\cap$ SR $\neq \varnothing$

= FALSE otherwise

Furthermore, let $\boldsymbol{S_{correct\ final}}$ denote a subscription function $\boldsymbol{S_{correct}}$ after all update and subscription region modifications over time during the simulation execution, that is for $\boldsymbol{MUR_{final}}$ and $\boldsymbol{MSR_{final}}$.

Lemma 1. (Correct generation of message sends and retractions in each step of DDM algorithm) Let AD_{final} be a sequence of ***Add*** and ***Delete*** operations issued by the algorithm for a particular region UR and federate F, and U be an Update operation at time T_U. Now, consider an ***Add*** or ***Delete*** operation that is defining the value of $\boldsymbol{S_{final}(F, UR, T_U)}$ to be either TRUE or FALSE. Such an operation has the largest time-stamp of all the ***Add*** and ***Delete*** operations from AD_{final} with time-stamps not greater than T_U. Consider the case when final ***Add*** or ***Delete*** operation appears after this Update. For this particular Update, algorithm generates sequence of send and retraction messages $U_S = <U_1, U_2 \ldots U_{final}>$ which satisfies the following properties:

1) If federate F is subscribed to receive updates from update region UR at time T_U, that is, if $\boldsymbol{S_{final}(F, UR, T_U)}$ = TRUE, then:
    - U_S is non-empty.
    - $U_1 = U_{final}$ = message send for this Update.
    - Each message send except U_{final} from the sequence is followed with a retraction of the same message.

2) Otherwise, if federate F is not subscribed to receive updates from update region UR at time T_U, that is, if $\boldsymbol{S_{final}(F, UR, T_U)}$ = FALSE, then:

   - U_S may be empty.
   - If U is non-empty, then U_1 = message send and U_{final} = message retraction for this Update.
   - Each message send from the sequence is followed with a retraction of the same message.

Proof. Proof is by induction. Essentially, we will prove that U_1 must be a message send for this Update, and that each message send from the sequence is followed with a retraction of the same message. Depending on the value of $\boldsymbol{S_{final}(F, UR, T_U)}$, U_{final} will either be a message send or a message retraction for this Update.

Initially, when the Update is issued, the AD sequence will be non-empty with any number of ***Add*** and ***Delete*** operations (note that the initialization sets a ***Delete*** operation at time $-\infty$). A message send will occur if $\boldsymbol{S(F, UR, T_U)}$ = TRUE, setting the sequence of send and retraction messages U_S to contain only this message send. Otherwise, no action will occur, leaving the sequence U_S empty. Thus, the lemma holds.

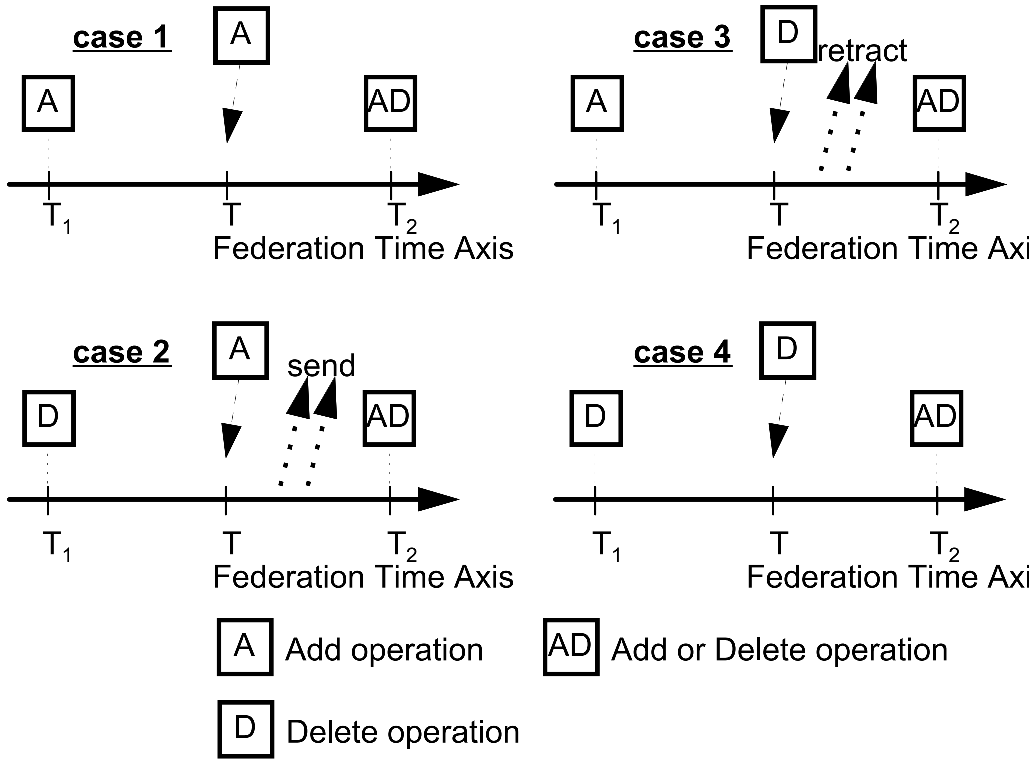


Figure 7 Possible cases when a new Add or Delete operation arrives.

Now we discuss an induction step. By definition, the "effect" of an Add or Delete operation only spans the interval from its time stamp to the time stamp of the immediately following ***Add*** or ***Delete*** operation. This is clearly implemented in our algorithm. If the time of an Update T we are considering is outside this interval, a new ***Add*** or ***Delete*** operation will not affect ***S(F, UR, T_U)*** after the operation has completed, nor it will change sequence of send and retraction messages U_S. Consequently, the induction step preserves the lemma in this case.

Next we consider the case where the time of an Update T_U is inside the interval affected by a new ***Add*** or ***Delete*** operation. Let us consider an Update with time stamp in the interval [T, T_2), where T_2 is the time stamp of the following ***Add*** / ***Delete*** operation, while T_1 is the time stamp of the preceding ***Add*** / ***Delete*** operation. Since the behavior of our algorithm depends on the new ***Add*** or ***Delete*** operation and the preceding ***Add*** / ***Delete*** operation, there are four possible cases depicted in Figure 7 which can be executed by the algorithm.

Let us first consider the situation where ***Add*** is a new operation from sequence AD defining the value of ***S*** to be TRUE. Case 1 corresponds to a new Add operation where the federate is already subscribed, that is S at time T_U is TRUE. This situation of two consecutive (in federate time) Add operations could arise if the Delete operation for the earlier Add operation had been delayed. Therefore, no further action is required other than noting that the Add operation has occurred. Case 2 corresponds to the situation where the federate is not subscribed, that is S at time T_U is FALSE. The federate should, but has not yet received any Updates with time stamp in the interval [T, T_2), where T_2 is the time stamp of the following ***Add*** / ***Delete*** operation, so messages for these updates, including the one for Update at T_U, must be sent to the federate. Updates with time stamp larger than T_2 have already been correctly processed by the operation at time T_2.

Now consider ***Delete*** operations. Case 3 corresponds to a Delete operation when the federate is subscribed. In this case, the federate has been sent messages with time stamp in the interval [T, T_2), where T_2 is the time stamp of the following ***Add*** / ***Delete*** operation, so these messages, including the one for Update at T_U, must be retracted (canceled). Case 4 corresponds to a Delete operation occurring when the federate is not subscribed to receive updates. Like case 1 for the Add operation, no additional messages (or retractions) need to be sent.

In all four cases, the induction step preserves the stipulations of the lemma, and hence the lemma is proven.

Def 6. (DDM Algorithm Correctness) Let $\boldsymbol{MSR_{final}}$ and $\boldsymbol{MUR_{final}}$ be the ***MR*** sets of all subscription and update region modifications respectively over time during the simulation execution. The DDM algorithm is correct if the following is satisfied.

1) If federate F is subscribed to receive updates from update region UR at time T, that is, if $\boldsymbol{S_{final\ correct}(F, UR, T)}$ = TRUE, then:
    - Update @ T should be delivered once and only once to federate F.

2) Otherwise, if federate F is not subscribed to receive updates from update region UR at time T, that is, if $\boldsymbol{S_{final\ correct}(F, UR, T)}$ = FALSE, then:
    - Update @ T should not be delivered to federate F.

Theorem 1. (DDM Algorithm Correctness) The time managed DDM algorithm 2 is correct according to Definition 6.

Proof. According to Theorem 2, the algorithm yields a subscription function $\boldsymbol{S_{final}}$ which satisfies $\boldsymbol{S_{final} = S_{final\ correct}}$. According to Def. 4 of $\boldsymbol{S_{final}}$ this means that the final ***Add*** and

Delete operations were also correctly issued. Depending on the value of $\boldsymbol{S_{final}}$, two situations are possible.

Let us first consider the situation where $\boldsymbol{S_{final}}$ is TRUE. Now, consider an ***Add*** or ***Delete*** operation that is defining the value of $\boldsymbol{S_{final}}$ to be either TRUE or FALSE. Such an operation has the largest time-stamp of all the ***Add*** and ***Delete*** operations with time-stamps not greater than T. In our case, since $\boldsymbol{S_{final}}$ is TRUE, such an operation is ***Add***. Now we consider two sub-cases depending on the ordering of this ***Add*** operation and Update at time T. When the ***Add*** appears before the Update, the distribution list for update region UR is adjusted to include federate F. Changing a distribution list occurs only at the beginning of an ***Add*** or ***Delete*** operation and since this ***Add*** operation defines $\boldsymbol{S_{final}}$ at time T, no additional ***Add*** or ***Delete*** operations can remove federate F from the distribution list at time T. Thus, when the Update is issued, it will be sent to federate F, and no duplicates of this Update will ever be sent again.

The second sub-case is when the ***Add*** appears after the Update. However, before a final ***Add***, there may be a sequence of ***Add*** and ***Delete*** operations which can cause state ***S*** to alternate between TRUE and FALSE at time T. This is apparent from the first line of ***Add*** and ***Delete*** operations, which either adds or deletes federate F from a distribution list D(UR) at time T. According to Lemma 1, an Update at time T may be sent and retracted multiple times until the final ***Add*** occurs. Furthermore, message sends for the same Update at time T will never follow each other, and retraction can only follow an Update send. Lemma 1 shows that after the final ***Add***, there is only one sent and non-retracted Update at time T, while other possible sends of this Update have been retracted. As we discussed before, since this ***Add*** operation defines $\boldsymbol{S_{final}}$ at time T, no additional ***Add*** or ***Delete*** operations can remove federate F from the distribution list at time T, and consequently cause another retraction. Also, no duplicates of this Update will ever be sent again.

We explain in a similar manner the situation where $\boldsymbol{S_{final}}$ is FALSE. Now, consider an ***Add*** or ***Delete*** operation that is defining the value of $\boldsymbol{S_{final}}$ to be either TRUE or FALSE. Such an operation has the largest time-stamp of all the ***Add*** and ***Delete*** operations with time-stamps not greater than T. In our case, since $\boldsymbol{S_{final}}$ is FALSE, such an operation is ***Delete***. Now we consider two sub-cases depending on the ordering of this ***Delete*** operation and Update at time T. When the ***Delete*** appears before the Update, the distribution list for update region UR is adjusted if necessary not to include federate F. The changing of a distribution list occurs only at the beginning of an ***Add*** or ***Delete*** operation and since this ***Delete*** operation defines $\boldsymbol{S_{final}}$ at time T, no additional ***Add*** or ***Delete*** operations can add federate F to the distribution list at time T. Thus, when the Update is issued, it will not be sent to federate F, nor it will ever be sent later during the simulation execution.

The second sub-case is when the ***Delete*** appears after the Update. As we already noted before, before a final ***Delete***, there may be a sequence of ***Add*** and ***Delete*** operations which can cause state $\boldsymbol{S}$ to alternate between TRUE and FALSE at time T. This is apparent from the first line of ***Add*** and ***Delete*** operations, which either adds or deletes federate F from a distribution list D(UR) at time T. According to Lemma 1, the Update at time T may be sent and retracted multiple times until the final ***Delete*** occurs. Message sends for the same Update at time T will never follow each other, and retraction can only follow an Update send. Lemma 1 shows that after the final ***Delete***, either each Update message sent at time T has its corresponding retraction, or that this Update has never been sent. As we discussed before, since this ***Delete*** operation defines $\boldsymbol{S_{final}}$ at time T, no additional ***Add*** or ***Delete*** operations can add federate F to the distribution list at time T, and consequently cause another message send. Hence, the Update will never be delivered to the federate F. This concludes the proof of Theorem 1.

Before giving the proof of Theorem 2, that is, that our algorithm produces a unique subscription function $\boldsymbol{S_{final}}$ which satisfies $\boldsymbol{S_{final}} = \boldsymbol{S_{final\ correct}}$, we present three lemmas which show invariant properties of the time-managed DDM algorithm.

Lemma 2. (Counter C_{sbsc}[F]: strength@t holds the grid cell's subscription strength per federate) The algorithm's counter C_{sbsc} always holds a value equal to the number of subscription regions of a federate overlapping the cell represented by this counter, that is, according to Def. C1.

Proof. Proof is by induction. Initially, before issuing any MSR or MUR operations, C_{sbsc} is initialized by the algorithm to zero, which is the correct value, since there are no subscription regions covering any of the cells.

Now we discuss an induction step. C_{sbsc} is correct and we want to prove that after an MSR or MUR operation C_{sbsc}‘ is still correct. Clearly, the only operation affecting C_{sbsc} is

MSR, and hence MUR does not have to be considered. According to semantics of MSR and MUR operations region change takes effect in a time interval [T_{start}, T_{end}), where T_{start} is modify region's parameter indicating when the change occurs, and T_{end} is time of this region's next change in logical time or infinity, if such change does not exist. Hence, the correctness of C_{sbsc} is trivial for any logical time outside this interval since the MSR operation affects only C_{sbsc} values in the interval.

We focus now on proving by contradiction that C_{sbsc} is correct in the interval [T_{start}, T_{end}). Suppose C_{sbsc}' is not correct for a certain cell C, that is, after MSR operation C_{sbsc}' is not equal to the number of subscription regions from federate F overlapping cell C. There are four possible cases depending on whether or not a particular cell is overlapped by a subscription region SR and its next logical time change at T_{end}, that is SR'.

1. Case 1: SR does not overlap cell C and SR' does not overlap cell C. Clearly, the number of subscription regions from federate F overlapping cell C does not change. Cell C is neither a new nor an old cell. According to the MSR operation, only C_{sbsc} of the new and old cells are updated, leaving C_{sbsc}' unchanged, and hence still equal to the number of subscription regions. This is a contradiction.

2. Case 2: SR does not overlap cell C and SR' overlaps cell C. Clearly, the number of subscription regions from federate F overlapping cell C is increased by 1. According to the MSR operation, cell C is a new cell and C_{sbsc} is incremented by 1. Hence, C_{sbsc}' is equal to the number of subscription regions. This is a contradiction.

3. Case 3: SR overlaps cell C and SR' does not overlap cell C. Clearly, the number of subscription regions from federate F overlapping cell C is decreased by 1. According to the MSR operation, cell C is an old cell and C_{sbsc} is decreased by 1. Hence, C_{sbsc}' is equal to the number of subscription regions. This is a contradiction.

4. Case 4: SR overlaps cell C and SR' overlaps cell C. Clearly, the number of subscription regions from federate F overlapping cell C remains the same. Cell C is neither a new nor an old cell. According to the MSR operation, only C_{sbsc} of the new and old cells are updated, leaving C_{sbsc}' unchanged, and hence still equal to the number of subscription regions. This is a contradiction.

After exhausting all the possible cases we can conclude that C_{sbsc}' is indeed correct for every cell. This concludes the proof of this lemma.

Lemma 3. (Counter $U_{cumulative}$[UR][F]: strength@t holds the cumulative subscription strength per update region) The algorithm's counter **$U_{cumulative}$** always holds a value equal to a sum of subscription strengths for this region of the grid cells overlapping an update region, that is, according to Def. C2.

Proof. Proof is by induction. Initially, before issuing any MSR or MUR operations, $U_{cumulative}$ is initialized by the algorithm to zero, which is the correct value, since this update region doesn't have any instance yet, or in other words it does not cover any of the cells.

Now we discuss an induction step. $U_{cumulative}$ is correct and we want to prove that after an MSR or MUR operation $U_{cumulative}$' is still correct. According to the semantics of the MSR and MUR operations, region changes take effect in a time interval $[T_{start}, T_{end})$, where T_{start} is modify region's parameter indicating when the change occurs, and T_{end} is time of this region's next change in logical time or infinity, if such change does not exist. Hence, correctness of $U_{cumulative}$ is trivial for any logical time outside this interval since MSR and MUR operations affect only $U_{cumulative}$ values in the interval.

MUR operation clearly updates $U_{cumulative}$ according to Def. C2. First assignment in Phase 1:

$$U_{cumulative}[UR][F]: strength@t = \sum_{\forall \text{ overlapping cells at time } t} C_{sbsc}[F]: strength@t$$

represents Def. C2 itself. The second assignment which is actually used by the algorithm is intended to achieve speedup of this update by considering only new and old cells:

$$\begin{aligned} U_{cumulative}[UR][F]&: strength@t \\ &= \sum_{\forall \text{ overlapping cells from previous instance of } SR} C_{sbsc}[F]: strength@t \quad \text{// speed up with incremental update} \\ &+ \sum_{\forall \text{ new cells}} C_{sbsc}[F]: strength@t \; - \sum_{\forall \text{ old cells}} C_{sbsc}[F]: strength@t \end{aligned}$$

Note that the first sum is the previous value of $U_{cumulative}$ and as such it does not need to be recomputed. Clearly this is correct since all overlapping cells of the new region's change are all the cells from the region's previous instance excluding those cells that are no longer overlapped (i.e. old cells) and including newly covered cells (i.e. new cells). This proves the correctness of updating $U_{cumulative}$.

Similarly, the MSR operation clearly updates $U_{cumulative}$ according to Def. C2. This is evident from Phase 2 :

$$U_{cumulative}[UR][F]: strength@t = \sum_{\forall \text{ overlapping cells at time } t} C_{sbsc}[F]: strength@t$$

$$= \sum_{\forall \text{ overlapping cells from previous instance of SR}} C_{sbsc}[F]: strength@t \quad \text{// speed up with incremental update}$$

$$+ \sum_{\forall \text{ new overlapping cells}} C_{sbsc}[F]: strength@t \; - \sum_{\forall \text{ old overlapping cells}} C_{sbsc}[F]: strength@t$$

Lemma 4. (Equivalence of the cumulative subscription strength per update region $U_{cumulative}$[UR][F]: strength@t and the subscription function S) The algorithm updates $U_{cumulative}$ in such a way that the following two properties are satisfied:

1. $U_{cumulative} = 0 \;\Leftrightarrow\; S = \text{FALSE}$
2. $U_{cumulative} \neq 0 \;\Leftrightarrow\; S = \text{TRUE}$

Proof. This can also be proved by induction. Initially, before issuing any MSR or MUR operations, $U_{cumulative}$ is initialized by the algorithm to zero, which is the correct value, since this update region doesn't yet have any instance, or in other words it does not cover any of the cells. Function S is FALSE initially, so that properties 1 and 2 are satisfied.

Now we discuss the induction step. Function S can only be changed by generating new ***Add*** and ***Delete*** operations. The only places in the algorithm where this happens are phases Phase 3 and Phase 2 of MSR and MUR operations, respectively. It is clear that generating ***Add*** and ***Delete*** operations happens only for update regions for which $U_{\text{cumulative}}$ has changed. Furthermore, for each of these update regions, an ***Add*** is generated only if $U_{\text{cumulative}}$ changes its value from 0 to greater than 0, implying that S becomes TRUE. Similarly, a ***Delete*** is generated only if $U_{\text{cumulative}}$ changes its value greater than 0 to 0, implying that S becomes FALSE. For all logical times not affected, properties 1 and 2 stand by the assumption of the induction step.

Theorem 2. (Equality of subscription functions*:* $\boldsymbol{S_{final} = S_{final\ correct}}$**)** For given sets $\boldsymbol{MSR} \subseteq \boldsymbol{MSR_{final}}$ and $\boldsymbol{MUR} \subseteq \boldsymbol{MUR_{final}}$ of subscription and update region modifications respectively over time during the simulation execution, the algorithm produces $\boldsymbol{S}$ which satisfies the following properties:

- The value of $\boldsymbol{S}$ is always the same as $\boldsymbol{S_{correct}}$, that is $\boldsymbol{S = S_{correct}}$.
- Consequently, final values are also the same, that is $\boldsymbol{S_{final} = S_{final\ correct}}$.

Proof. This theorem can be proved by induction. Initially, before issuing any subscription or update region modifications $\boldsymbol{S = S_{correct}}$ = FALSE. According to Theorem 3, subscription function $\boldsymbol{S}$ is updated correctly in each iteration of the algorithm, resulting in $\boldsymbol{S = S_{correct}}$. By induction over all subscription or update region modifications this will be the case at the end of the simulation as well. So we have $\boldsymbol{S_{final} = S_{final\ correct}}$, which proves the theorem.

Observation. Subscription function $\boldsymbol{S_{correct}}$, and hence $\boldsymbol{S}$ never depends on the order in which subscription and update region modifications are issued. This follows directly from the definition of correct subscription function $\boldsymbol{S_{correct}}$ and Theorem 2.

Theorem 3. (Subscription function *S* is updated correctly in each iteration of the algorithm) Let $\boldsymbol{MSR \subseteq MSR_{final}}$ and $\boldsymbol{MUR \subseteq MUR_{final}}$ be sets of subscription and update region modifications respectively over time during the simulation execution. Also let $\boldsymbol{m}$ be the next modification, that is $\boldsymbol{m \in MSR_{final} \setminus (MSR \cup MUR)}$**.** If the current state of the algorithm satisfies $\boldsymbol{S = S_{correct}}$ for every update region, then the next iteration of the algorithm that processes modification $\boldsymbol{m}$ produces subscription function $\boldsymbol{S}$ which still satisfies property $\boldsymbol{S = S_{correct}}$.

Proof. Proof is by contradiction. Suppose $S' \neq S_{correct}'$ for particular update region. There are four possible cases depending on the values before an MSR or MUR operations S and $S_{correct}$, and values after these operations S' and $S_{correct}'$ for this update region.

1. Case 1: $S = FALSE \quad S_{correct} = FALSE$

   $S' = TRUE \quad S_{correct}' = FALSE$

By Lemma 4, S' = TRUE implies that $U_{cumulative}' \neq 0$.

On the other hand, by Lemma 3, $U_{cumulative}$ holds the cumulative subscription strength per update region, according to Def. C2, that is:

$$U_{cumulative}` = \sum C_{sbsc}`$$
$$\forall \textit{overlapping cells at time } t$$

By Lemma 2, C_{sbsc} holds a grid cell's subscription strength per federate, that is, according to Def. C1. Thus, the same holds for C_{sbsc}'. Considering an assumption that $S_{correct}$' = FALSE for update region, this further implies that C_{sbsc}' = 0 for all cells overlapped by the region. Hence, $U_{cumulative}$' = 0, which is a contradiction.

2. Case 2: S = FALSE $S_{correct}$ = FALSE

   S' = FALSE $S_{correct}$' = TRUE

By Lemma 4, S' = FALSE implies that $U_{cumulative}$' = 0.

On the other hand, by Lemma 3, $U_{cumulative}$ holds the cumulative subscription strength per update region, according to Def. C2, that is:

$$U_{cumulative}` = \sum C_{sbsc}`$$
$$\forall \textit{overlapping cells at time } t$$

By Lemma 2, C_{sbsc} holds a grid cell's subscription strength per federate, that is, according to Def. C1. Thus, the same holds for C_{sbsc}'. Considering an assumption that $S_{correct}$' = TRUE for update region, this further implies that C_{sbsc}' $\neq 0$ for at least one cell overlapped by the region. Hence, $U_{cumulative}$' $\neq 0$, which is a contradiction.

3. Case 3: S = TRUE $S_{correct}$ = TRUE

   S' = FALSE $S_{correct}$' = TRUE

By Lemma 4, S’ = FALSE implies that $U_{cumulative}$‘ = 0.

On the other hand, by Lemma 3, $U_{cumulative}$ holds the cumulative subscription strength per update region, according to Def. C2, that is:

$$U_{cumulative}` = \sum C_{sbsc}` \quad \forall \text{ overlapping cells at time } t$$

By Lemma 2, C_{sbsc} holds a grid cell’s subscription strength per federate, that is, according to Def. C1. Thus, the same holds for C_{sbsc}‘. Considering an assumption that $S_{correct}$’ = TRUE for update region, this further implies that C_{sbsc}‘ $\neq 0$ for at least one cell overlapped by the region. Hence, $U_{cumulative}$‘ $\neq 0$, which is a contradiction.

4. Case 4: S = TRUE $S_{correct}$ = TRUE

    S’ = TRUE $S_{correct}$’ = FALSE

By Lemma 4, S’ = TRUE implies that $U_{cumulative}$‘ $\neq 0$.

On the other hand, by Lemma 3, $U_{cumulative}$ holds the cumulative subscription strength per update region, according to Def. C2, that is:

$$U_{cumulative}` = \sum C_{sbsc}` \quad \forall \text{ overlapping cells at time } t$$

By Lemma 2, C_{sbsc} holds a grid cell’s subscription strength per federate, that is, according to Def. C1. Thus, the same holds for C_{sbsc}‘. Considering an assumption that $S_{correct}$’ =

FALSE for update region, this further implies that C_{sbsc}‘ = 0 for all cells overlapped by the region. Hence, $U_{cumulative}$‘ = 0, which is a contradiction.

Consequently, S’ ≠ $S_{correct}$’.

3.5 Design Rationale

We now summarize the design decisions that were made in the time-managed DDM algorithm. The following sections cover the design rationale and tradeoffs for dealing with missed messages and messages in the past, as well as how to represent distribution lists and how to match update and subscriptions regions.

3.5.1 Dealing with Missed Messages and Messages in the Past

The log-based approach described earlier, in section 3.3, solves the problem of missed messages and messages in the past. Update messages are logged as they are issued. When late subscriptions change connectivity between logical processes and indicate that some updates haven’t been sent to certain LPs, messages are retrieved from the log and resent to those LPs. To avoid messages in the LP’s past, time advance operations (e.g. LBTS) need to be modified to account for the messages that are resent from the logs after such operations have been initiated. For example, late subscriptions that cause messages to be resent from the log after an LBTS computation has been initiated will not result in these messages being received in the LP’s past. Such late messages will properly be accounted for in the modified LBTS computation.

An alternative to dealing with the missed messages and messages in the past is to have an optimistic technique with general state saving and rollback and recovery mechanisms. When the system detects missed messages or messages in the past, it rolls back its state to a previously saved safe state, and sends anti-messages for the messages that need to be canceled. Two optimistic techniques are described in section 3.3.

Optimistic techniques have an advantage of being readily available and optimized for different types of simulations. However, there is a substantial performance cost associated with the general purpose rollback mechanisms: state saving and message cancellation. Our log-based approach doesn't have such costs, since it is able to recover from errors by exploring the semantics of DDM operations. The second benefit is that optimistic techniques are not applicable to all types of simulation. For example, in a training simulation with human participants, rolling back to a previously saved state is inappropriate. State of the simulated world can only be seen as advancing forward in time when human participants are involved.

3.5.2 Distribution List Representation

Distribution lists are at the core of the distribution lists layer (Figure 5). They keep track of LPs that are subscribed to receive updates at different logical times, and this information evolves over the simulation execution. Multiple versions of distribution lists are needed corresponding to different logical times.

One way to realize distribution lists is to use space-time memory and unicast network communication. An abstraction called space-time memory can be used to keep time evolving data such as our distribution lists [31, 45]. Space-time memory is similar to ordinary memory except a time stamp is specified each time the data structure is read or modified. A read operation returns the most recent version of the data structure as of the time stamp of the read, providing a convenient means for accessing the correct version. Thus, when an LP issues an update at logical time T, the space-time memory is read to determine a distribution list at time T, followed by sending (i.e. unicasting) an update message to all LPs in the list. The drawback of this approach is that it does not utilize the network multicast capability that can reduce message delays in the network and allow for more messages until the network reaches its capacity and becomes the bottleneck.

Another way to realize distribution lists is completely within the network. A multicast group is assigned for a distribution list at every logical time when the list changes to include a new LP or exclude an LP which is currently a member of the distribution list. This approach is viable only for simulations with small number of distribution lists, and where the lists evolve infrequently. Otherwise, the problem arises when there are not enough multicast groups to handle all the different versions of distribution lists. Currently, networking hardware capabilities are still limited so that the physical multicast groups can be viewed as a scarce resource. Furthermore, there is a significant overhead in using a physical multicast group. This overhead is manifested as the delay from the time when multicast operation is issued (e.g. JOIN or LEAVE), until the network changes its state to reflect this operation.

The multicast realization of our algorithm is based on the space-time memory and limited number of multicast groups, one for each update region. Multiple versions of the distribution list cannot be easily used because the lists are represented within the network infrastructure by membership to multicast groups. This problem can be solved by including in the multicast groups the union of all distribution lists that are currently active. Add operations result in immediate joins to the multicast group, if an LP is not already joined to the group. Delete operations are delayed until the memory reclamation mechanism advances to the time of the operation, and no other Add operations exist in the future.

The advantage of this approach is that the number of multicast groups that are required depends only on the number of update regions for all the LPs in the simulation. The required number of groups does not depend on how frequently the distribution lists evolve, as was the case when the lists were realized entirely within the network. Hence, our approach is not limited to simulations with only infrequent multicast group membership changes. On the other hand, the drawback is that there will be extra messages due to the one multicast group per update region's time line design. It was explained that an LP may be joined to a multicast group even if that LP is not subscribed (i.e. deleted) at some logical time intervals from its current time into the future. This could result in extra update messages that have to be discarded at the destination LPs. Our approach can be enhanced with more multicast groups per update regions' timeline, by dedicating a multicast group per a distinct portion of the timeline. This topic can be explored further as future research.

3.5.3 Matching of Update and Subscriptions Regions

The interest management layer (Figure 5) is responsible for matching of update and subscription regions in order to determine Add and Delete operations for the Distribution list layer. Our hybrid approach described in detail in sections 3.3.2 and 3.4 partitions the routing space into grid cells, which are used to determine update and subscription regions' overlaps. Two counters are defined for this purpose, and space-time memory is used to capture counter changes over time. A cell counter keeps track of how many subscription regions of an LP overlap a cell at a particular logical time. A region counter is a sum of cell counters for all grid cells overlapped by an update region. The interest management layer generates an Add operation when region counter changes its value from zero to a value greater or equal to 1. Conversely, a Delete operation is generated when the region counter decreases its value to zero.

This approach is faster than the pure region-based approach, where every region change entails comparing that region with every other region to determine possible overlaps. In our approach, we limit searching only to grid cells overlapped by a region, and hence only the regions overlapping those grid cells need to be considered. More precisely, when a region moves over logical time, we don't need to update cell counters for common cells (i.e. cells overlapped by old and new region's instance), nor do we have to consider common cells for updating region counters. This implies that when regions move gradually over the routing space, only the outermost cells that define region's circumference have to be checked for updating cell and region counters.

On the other hand, the region-based approach yields a perfect matching, while the preciseness of the hybrid approach depends on the chosen cell size relative to the region sizes. In fact, the hybrid approach also yields a perfect matching when regions always overlap grid cells fully. In general, the hybrid approach is less efficient but more precise as the cell size decreases since there are more cells to consider, and vice versa. Section 2.4.2 discussed the research that compares the cost of region and grid-based approaches and how to determine the optimal grid cell size. Our approach can easily be extended to cover the perfect matching, by defining two additional cell and region counters to account for cells that are not fully overlapped. This is one of the paths to explore in the future synchronized DDM implementations.

The pure grid-based approach, where a unique multicast group is assigned to all grid cells, has a few shortcomings in comparison to the hybrid approach. Distribution lists are represented implicitly as multicast groups within the network, and there is no computational overhead to determine region overlaps. A JOIN operation is issued for cells that region overlaps, while a LEAVE operation is issued when a cell is not overlapped after an object moves to a new portion of the routing space. The problem is that during an update operation, an update message has to be sent to all the multicast groups overlapped by the corresponding update region. This will generate multiple copies of the same message destined for an LP, unless an update region covers only one grid cell. Considering that update operations are more frequent than region changes, a pure grid-based approach is very inefficient. Furthermore, it is limited to the case when regions overlap grid cells fully.

Finally, matching approaches with varied cell sizes and different levels of accuracy have been proposed. For example, an approach that uses multidimensional binary trees [46] is well suited for matching in unsynchronized DDM systems. Different portions of the routing space may in effect be assigned varied cell sizes, which is beneficial for the matching speed when region sizes vary largely. Larger cells are assigned to a portion of the routing space with larger regions, and vice versa. This research direction for synchronized DDM systems is open for future investigation.

CHAPTER 4

PERFORMANCE EVALUATION

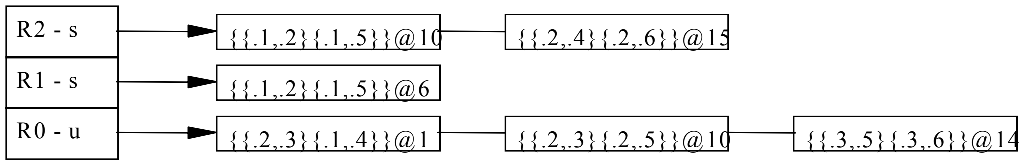
4.1 Implementation

4.1.1 Simplified Space-Time Memory

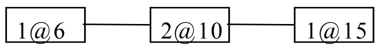
Multiple versions of different data structures corresponding to different logical times are needed. As discussed in the previous chapter, our approach to realizing distribution lists is by using the space-time memory mechanism for time evolving data and multicast groups for each update region. Space-time memory is a two dimensional memory system that is addressed using both spatial and temporal coordinates. Space-time memory has been proposed as an efficient implementation of Virtual Time Memory (VTM) architecture [31]. VTM is an advanced computer architecture that detects data dependency violations at runtime, and automatically recovers by using rollbacks. Another use of space-time memory is as a high level data sharing abstraction for threads in the Stampede project [45]. It has a complex design that subsumes buffer management, inter-task synchronization, meeting soft real-time constraints, etc.

In our simplified implementation we are using doubly linked lists, sorted by version time stamps. The data structures each federate must maintain are shown in Figure 8.

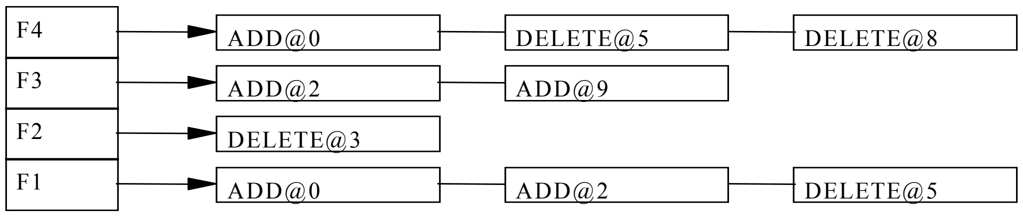
There is a multiple version data structure for each of the regions ("u/s" denotes update/subscription regions) defined by the federate. Each element represents a set of extents for the region at a time given by the time stamp. Each of the cells in the routing space has a counter with its associated versions, as well as logs for Add/Delete operations, received from other federates. Instead of having one log, we have a log for every federate. This will speed up recording a new Add/Delete entry from a federate by not having to traverse log elements of other federates. More importantly, it will speed up finding a distribution list at a given time t, by not having to traverse irrelevant elements from some federates (i.e. elements whose time stamps are less than t). This is illustrated in the following example.



Regions and corresponding versions of extents.



Versions of a counter, used by the IM2 sublayer.



Federates and corresponding versions of ADD/DELETE operations, used by the DL layer.

Data structures for every cell in each of the routing space.

Figure 8 Data structures for federate:F0.

An UpdateAttributeValue@7 issued for an attribute associated with an update region that is equal to a cell depicted above causes a backward search through the Add/Delete logs. Traversed elements are: F4:8,5;F3:9,2;F2:3;F1:5. The distribution list for this update is {F3}, and a message containing the update is being sent to F3. Note that in general, we could have many log elements for F4 between simulation times 2 and 5 that would have to be searched in the case of a unique log, but this is not the case for this example.

4.1.2 Distributing the DDM Algorithm's Data Structures

The various data structures that are required to implement DDM may be centralized, or distributed among the processors participating in the federation execution.

In our implementation we realize matching of update and subscription regions by distributing data. Conversely, matching can also be realized by a centralized coordinator [6], which can be seen as keeping the connection information database between LPs [9]. However, this approach is not efficient as will be illustrated in the following. To simplify the discussion, let us assume the connection database is mapped to a single logical process, called the connection database logical process (CDLP). Each service invocation that must access connectivity information must send a message to the CDLP with time stamp equal to that used in the service invocation. When the CDLP processes this message, it reads and/or modifies the database according to the type of operation that is required, and completes the realization of the operation.

The problem with the centralized coordinator is that even when region changes and update operations have non-zero lookaheads, receiving such events at the CDLP may generate resend and retraction events with the same time-stamp, that are in effect zero-lookahead events. Thus, the centralized coordinator becomes the bottleneck since it enforces sequential execution of the DDM events. Our optimistic algorithm to time-managed DDM doesn’t have such a problem.

In a distributed implementation, the data structures may be replicated to enable fast lookup, at the expense of additional communication to keep the multiple copies consistent. Our implementation of DDM uses a replicated copy of the data structures in each processor. This is justified by an assumption that region changes are less frequent operations than updates.

In particular, each subscription region change associated with attributes is sent to every federate doing updates for any of these attributes. When region change information is received, grid and cumulative data structures are being updated according to our algorithm. Optimizations are possible to avoid sending information concerning every subscription region change. If a subscription region covers cells already covered by other subscription regions in a time interval that is active, the subscribing federate doesn’t need to send this information since there will not be any change in distribution lists or group memberships. However, after other subscription region changes, it may turn out that this information was relevant, and appropriate data will be sent to publishers at that time.

4.1.3 Implementing Message Resends and Retracts

Late subscriptions (i.e. late subscription region changes) or late update region changes can cause changes in Add / Delete logs. In the worst case, each Add operation that was previously issued starting at an LP's current time will be cancelled (e.g. replaced) with a Delete operation, and vice versa. Canceling an operation causes update messages to be resent from the logs, as was discussed in the previous chapter.

Our implementation treats canceling operations resulting from a single region change independent from each other. Hence, instead of resending or retracting an update only once, it may be resent or retracted multiple times, up to the maximum number of times which equals the number of Add / Delete operations that need canceling. Optimized implementation, on the other hand, would resend or retract updates only once per region change. The impact on performance is that our implementation may generate more message resends and retracts sooner (e.g. for lower lookahead values) than the corresponding optimized implementation. The difference in implementation efficiency is analyzed later in section 4.5.

4.1.4 Georgia Tech's FDK Overview

Georgia Tech's FDK software [43] is being used for our experiments, which is designed to facilitate building efficient run-time infrastructures (RTIs) that can be used to federate simulations. Federated simulations, in which different simulators interoperate with each other in executing a single simulation are increasingly becoming important in many areas, such as battlefield simulation (e.g. SIMNET[21], JPSD[24]) and distributed multi-user games (e.g. DIVE[44]).

FDK stands for the Federated Simulations Development Kit package. It is being used in a variety of educational and research projects such as research in DDM, use of high bandwidth and active networks for distributed simulations, and federated simulations for modeling telecommunication networks (e.g. Parallel and Distributed NS (PDNS)[41] and the Georgia Tech Network Simulator (GTNetS)[42]). RTI-Kit is the simulation engine component of the FDK.

RTI-Kit is a collection of libraries designed to support development of Run-Time Infrastructures (RTIs) for parallel and distributed simulation systems. Each library can be used separately, or together with other RTI-Kit libraries, depending on what functionality is required. These libraries can be embedded into existing RTIs, e.g., to add new functionality or to enhance performance by exploiting the capabilities of a high performance interconnect. The RTI-Kit software was successfully embedded into an HLA RTI developed in the United Kingdom [32, 33]. Alternatively, the libraries can be used in the development of new RTIs.

This "library-of-libraries" approach to RTI development offers several important advantages. First, it enhances the modularity of the RTI software because each library within RTI-Kit is designed as a stand alone component that can be used in isolation of other modules. Modularity enhances maintainability of the software, and facilitates optimization of specific components (e.g., time management algorithms) while minimizing the impact of these changes on other parts of the RTI. This design approach facilitates technology transfer to other RTI development projects because utilizing RTI-Kit software is not an "all or nothing" proposition; one can extract modules such as the time management while ignoring other libraries.

Multiple implementations of the RTI-Kit software have been realized targeting different platforms. Specifically, the current implementation can be configured to execute over shared memory multiprocessors such as the SGI Origin, cluster computers such as workstations interconnected via a low latency Myrinet switch [34], to workstations interconnected over local or wide area networks using standard network protocols such as IP.

The architecture for RTI software constructed using RTI-Kit is shown in Figure 9. At the lowest level is the communication layer that provides basic message passing primitives. Communication services are defined in a module called FM-Lib. This communication layer software acts as a multiplexer to route messages to the appropriate module. The current implementation of FM-Lib implements reliable point-to-point communication. It uses an API based on the Illinois Fast Messages (FM) software [35] for its basic communication services, and provides only slightly enhanced services beyond those of FM.

Above the communication layer are modules that implement key functions required by the RTI. These modules form the heart of the RTI-Kit software. Specifically, *TM-Kit* is a library that implements distributed algorithms for realizing time management services. Similarly, *DDM-Kit* implements functionality required for data distribution management services. *MCAST* is a library that implements group communication services. Other libraries, not shown in Figure 9, provide utilities such as software for buffer and queue management.

Finally, the interface layer utilizes the primitive operations defined by these modules to implement a specific Application Program Interface (API) such as the HLA Interface Specification. The current RTI-Kit distribution includes an implementation of a subset of the HLA IFSpec (version 1.3).

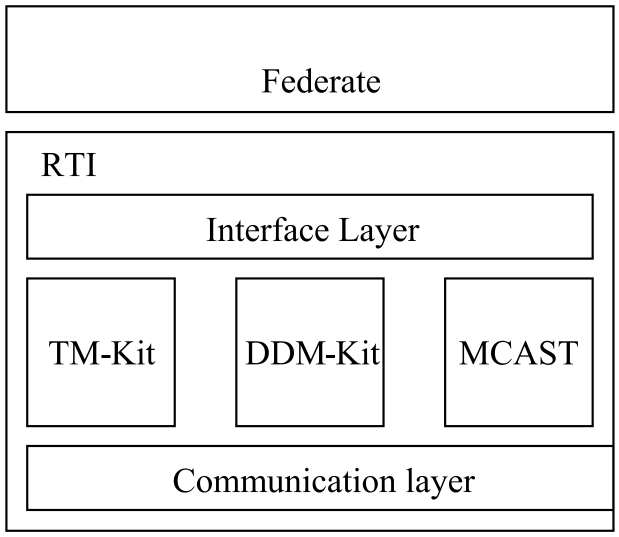


Figure 9 RTI architecture using RTI-Kit.

4.1.5 Hardware

All of the experiments were performed on a network of 8 Dell PowerEdge 2650 servers. These servers have dual Pentium 4 Xeon processors running at 2.8 GHz, and with 4 gigabytes of physical memory. The network used was a 100 Mbps Fast Ethernet.

4.2 Cost of Synchronization

Before presenting the performance results, we will characterize the costs of our time-managed DDM implementation.

- ***Message resends / retracts due to the late region changes.*** Late subscriptions (i.e. late subscription region changes) or late update region changes can cause messages to be resent or retracted from Add / Delete logs. The exact number of such messages depends on the simulation model's lookahead value and the benchmark application's characteristics such as the frequency of region changes and frequency of update operations (section 4.5). As will be seen shortly, this is one of the major contributing factors to the overall cost of the time-managed DDM.
- ***Extra messages during update operations due to the limited number of multicast groups.*** In order to represent multiple versions of a distribution list per update region with a single multicast group, we include the union of all versions of the distribution list that are currently active in that multicast group. Add operations result in immediate joins to the multicast group, if an LP is not already joined to the group. Delete operations are delayed until the memory reclamation mechanism advances to the time of the operation, and no other Add operations exist in the future. Hence, extra messages can occur when an update operation is issued at a logical time when an LP is joined but not really subscribed to receive updates, which is indicated by a Delete operation at that or an earlier logical time. These messages can simply be filtered out at the destination LPs. We will show that smaller lookahead values limit the amount of extra messages during the update

operations by limiting the concurrency. Furthermore, the number of these messages is much less than the number of message resends / retracts, and hence, such extra messages have less effect on performance.

- ***Extra messages due to the granularity of the grid partitioning.*** Grid cell size determines the level of accuracy for matching of update and subscription regions. Smaller grid cells result in better matching accuracy and fewer extra messages, but greater computational overhead to perform matching and larger memory requirements to keep time evolving data for more cells in the routing space. If grid cells are much larger than the regions, it is obvious that the extra message overhead will become more pronounce. At an extreme, when we have a single cell in the routing space, synchronized or unsynchronized DDM will in effect deteriorate into broadcast. In general, the cell size should not be larger than the smallest region's size in order to limit this kind of extra messages. Approaches exist [20] to determine optimal cell sizes for some applications. Our matching algorithm can easily be enhanced to completely avoid this problem, and the idea will be revisited in the last chapter.

4.3 Benchmark Applications

We utilize two applications that exercise different mobility patterns to test time-managed DDM, termed the Future Combat Systems (FCS) and the synthetic benchmark applications.

Both FCS and the synthetic benchmark application can be viewed as distributed simulations of moving entities across a simulated world, but with different mobility patterns. The entities are scattered over a grid of approximately 50 km by 50 km. Entities are simulated by federates that reside on different processors. Each entity is associated with a unique update and a subscription region. Update and subscription region size is varied, as described later in section 4.4.2. An entity's position changes and the logical time when the changes occur are specified in the mobility data file, which differ for each mobility pattern. Events are executed when the corresponding wall clock time (logical time + constant) is reached, with time-stamps assigned as the current logical time + lookahead.

Mobility data, that is how and when entities change their positions, is specific for each application and is contained in two files: nodes.in and mobility.in. These files specify mobility data for all the nodes. Furthermore, some entities move frequently, and others move rarely. The nodes.in gives the initial position of each entity. The format of the data is: nodenum 0 (x, y, z), i.e., object nodenum is at position (x, y, z) at logical time 0. The mobility.in describes all the entities' motion. The format of the data is: nodenum T (x, y, z), i.e. at logical time T, object nodenum moves to the new position (x, y, z). The timestamp units are in seconds.

Now we describe some unique characteristics of both benchmark applications.

- **Real World Benchmark Application - Future Combat Systems**

Future Combat Systems (FCS) is one of the benchmark applications for our synchronized DDM system. In this application, mobility data and logical time stamps are taken from a simulated demonstration exercise using the Joint Semi-Automated Forces (JSAF) simulation program.

The Future Combat Systems program (FCS) uses ad hoc wireless communications technology with the goal of bringing significant improvements in synchronization, data exchange, mobility and effectiveness of future military forces. It is envisioned as a scalable networked system of mobile systems comprised of both manned and unmanned platforms, involving land-based (e.g. troops, mobile equipment, robots), airborne (e.g. UAVs, satellites) and naval (e.g. ships, ground stations) assets. The movement of these units was scripted to model an engagement of mobile units with backup naval support to evaluate the effectiveness of wireless networking protocols in a typical FCS deployment. Traces of the movements of vehicles were collected and used to drive the distributed simulation experiments described here.

- **Synthetic Benchmark Application**

Our second benchmark application is also a distributed simulation of moving entities such as tanks or aircraft across a simulated world (routing space(s)). The path of each entity follows a random walk, with each entity equally likely to move in any new direction at each time. The distance between points as objects move is taken from an exponential distribution with the following means for each coordinate: Δx = 100 m and Δy = 100 m.

Logical time stamps for each object's new position (entries in nodes.in and mobility.in files) are also taken from an exponential distribution with mean of 200 seconds.

4.4 Experiments

Common parameters for all experiments include:

- The simulation consists of 8 logical processes (HLA federates), one per processor
- Each LP simulates 20 objects with distinct update and subscription regions, which totals 160 objects in the simulation
- An update is generated every 30 seconds of logical time
- Region changes and corresponding logical times happen according to the mobility patterns in FCS and synthetic applications
- Logical end time is 10000 seconds (or 2.8 hours of simulated time)

In order to understand what affects the performance of time-managed DDM, we vary these parameters for both types of mobility patterns (FCS and synthetic applications):

- Lookahead
- Region size
- Network buffer size, i.e. size of send and receive buffers

Performance data is shown as a function of lookahead for each set of experiments. The impact of varying the region size is presented in section 4.4.2, where we show what happens when regions are large so that they almost always overlap (section 4.4.2.1) as well as when they are of smaller sizes (section 4.4.2.2). On the other hand, the impact of network buffers is presented in section 4.4.3.

For each set of experiments we show the following measurements for both synchronized and unsynchronized DDM:

- Run time in seconds
- Run time confidence interval (95%) for time-managed DDM
- Number of message sends during Update operations
- Number of message resends / retracts due to interleaved Modify Region operations from different logical processes
- Number of multicast operations, that is, number of multicast joins and leaves

Presented measurements are obtained from a set of 100 experiments (except 50 experiments in section 4.4.3), each having the same input and run-time parameters. Measurements from all runs are taken at one logical process in the simulation, and then statistically analyzed. Gathered statistics such as mean values, etc. are shown for run time, number of message sends during Update operations, number of message resends and retracts, and number of issued multicast operations.

4.4.1 Impact of Lookahead on Performance

All of the performance measurements in this chapter are presented as a function of varied lookahead. Very small lookahead values (such as 1 second in these experiments) limit concurrency, and in effect cause all events, including DDM region change events to be serialized. In other words, region change events from different LPs are seen as occurring in time-stamp order at each logical process. Hence, for small enough lookaheads, synchronized and unsynchronized DDM are identical, and at the same time correct information about subscriptions at any particular logical time (assuming update and region change events are issued in time stamp order at every LP). This further implies that synchronized DDM does not need to resend or retract any messages. All of our experiments confirm that performance is essentially the same for time-managed and unsynchronized DDM for small lookahead values (such as 1 second).

Larger lookaheads, on the other hand, increase concurrency while reducing overall simulation execution overheads by requiring fewer LBTS computations to advance time. This can be seen in each set of experiments when run-time makes a dip around relatively small lookahead values. However, the trade-off is that there are an increasing number of messages that have to be resent and/or retracted from update logs due to out of time-stamp order receipt of region changes from different or even the same logical processes. This will be seen in the following sections.

4.4.2 Impact of Region Size on Performance

Each simulated entity is assigned a unique update and subscription region. Subscription region size is varied to span both "large" and "medium" regions, while update regions remain the same size. We choose two parameters to define update and subscription region size: update region diagonal distance and visibility range, respectively. Table 1 shows these parameters in meter units for large and medium size regions.

Table 1 Large and medium region size parameters

	Update region diagonal distance (meters)	*Visibility range (meters)*
Large region size	*1000*	*40000*
Medium region size	*1000*	*10000*

Like a geometric rectangle, an update region can be defined by an object's current position, the region's diagonal distance and a direction angle. An object's current position (e.g. x and y coordinates) and direction angle from which the object moves are always known during the execution from the mobility data. We just need the update region's diagonal distance to define the update region as a rectangular portion of the routing space.

On the other hand, we can define the subscription region by the lower left and upper right hand coordinate positions. Those two positions can easily be computed when the visibility range that defines how far the object can see is known. A subscription region's lower left position is simply $2^{½}$ * visibility distance away from the update region's lower left position, in a direction directly opposite to the object's movement. Similarly, a subscription region's upper right position is $2^{½}$ * visibility distance away from the update region's upper right position, in the direction of an object's movement.

The impact of region size on performance can be seen in the following two figures. Figure 10 shows side by side comparison of the run time of synchronized DDM for medium and large region sizes. Furthermore, the run time with the medium region sizes is shown for the FCS and synthetic mobility patterns, while the run time for the large region size is shown for the FCS mobility pattern. All three graphs depict similar initial behavior when performance improves due to increases in lookahead, as was explained in previous section. Increasing lookahead after a certain lookahead value has a negative impact on performance for medium region cases. This is due to the increases in the number of messages that need to be resent or retracted in order to fix errors (e.g. missed messages) that would otherwise occur in an unsynchronized DDM. Further increases in lookahead increase the run time even more, since greater lookahead allows more concurrency in the execution, which in turn causes more messages to be resent or retracted. Unsynchronized DDM, on the other hand, doesn't fix any DDM out of time stamp ordering errors. Hence, performance generally improves with larger lookahead values as shown in Figure 11. These effects will be discussed in detail in the following sections.

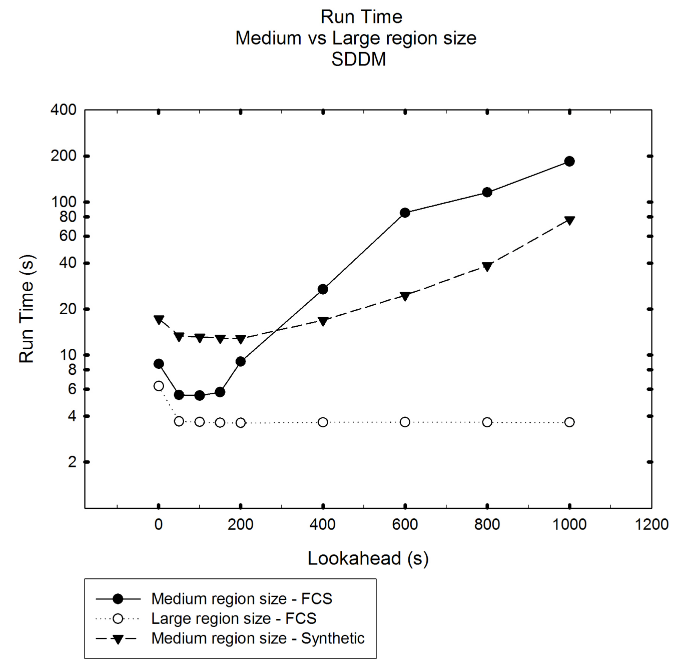


Figure 10 Run time when varying region size for SDDM.

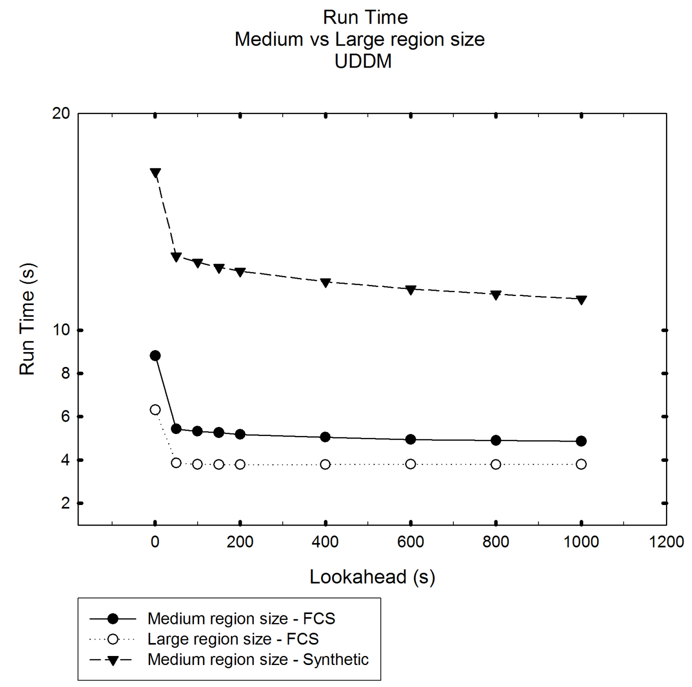


Figure 11 Run time when varying region size for UDDM.

When compared to unsynchronized DDM, it can be seen that the time managed DDM performs the same for all lookahead values when regions are large. The reason for this is that large regions cause connections between logical processes to be established when the simulation starts (i.e. all LPs are joined to every multicast group), and they never change during the simulation execution. On the other hand, the performance is the same only until a certain lookahead value for medium size regions. Specifically, performance of synchronized and unsynchronized DDM is similar until the lookahead reaches 150 seconds for the FCS mobility scenario, or 200 seconds for the synthetic scenario. The benefit of increasing lookahead outweighs the overhead of message resends and retracts until these lookahead values are reached. Further lookahead increases change this balance, when the run time of the time managed DDM increases an order of magnitude in comparison to the unsynchronized DDM case. Implementation optimizations are possible to reduce this difference in run time, as will be seen later in section 4.5.

4.4.2.1 Large Region Size

Large region size parameters (also shown earlier in Table 1) are:

- *Update region diagonal distance = 1000 meters*
- *Visibility range = 40000 meters*

Time-managed DDM services can be as efficient as unsynchronized DDM services when the mobility patterns cause infrequent group membership changes. This can be seen in the following experiments where region sizes are large, so they do not cause frequent group membership changes. Mobility patterns used are those from the FCS application. The same performance is expected from a random mobility patterns such as our synthetic application in this case.

As regions become larger in size, they cover larger portions of the routing space, and hence the probability increases that an object's update region overlaps another object's subscription region. At an extreme, regions that always cover the routing space completely cause both synchronized and unsynchronized DDM to behave like broadcast, and hence, perform the same as illustrated in Figure 12. Each update is sent to every logical process. The improvement in performance as lookahead is being increased from 1 second to 50 seconds was discussed in section 4.4.1. Furthermore, performance does not deteriorate for synchronized DDM, since there are no message resends / retracts, as will be explained below.

The difference in performance of unsynchronized and synchronized DDM in this case can be explained by the two implementation factors, that is by (1) different overheads during the matching phase of Modify Region operations and (2) different overheads during the multicast joins and leaves needed to complete these operations.

The first difference in overhead during the matching phase is a result of how time evolving data such as distribution list entries and region changes are stored. In the unsynchronized case, new data overwrites the old data. On the other hand, the synchronized implementation uses linked lists to capture changes instead of overwriting data. This results in a slight difference in performance during the matching phase which favors time managed DDM for the large regions in our implementation. This in turn causes a more noticeable difference during the multicast group changes that follow the matching phase.

Multicast join and leave operations have a major influence on timing of Modify Region operations for large regions. Cached copies of the distribution lists are kept at all processors, and this requires communication with all of them. A slight difference in overhead during the matching phase causes the timing of these multicast communication messages to be different, which ultimately results in more noticeable communication overheads, favoring the synchronized DDM in our implementation.

This broadcast-like behavior is apparent from the following figures. As expected, Figure 13 shows that each update is sent to every other LP. However, when the lookahead value is equal to 1 second, the number of message sends during the simulation for all updates is 46,760. This implies that each update is being broadcast (and received by other 7 logical processes), since there are 334 = upper bound (10000 seconds / 30 seconds) updates per object, and hence the total number of updates is 46,760 = 334 x 20 objects x 7 LPs. As for the lookahead value of 1 second, all updates are still sent to every other LP. The difference in the actual number of message sends is that the number of updates is just one less from all the other lookahead values (i.e. there are 333 updates per LP instead of 334, leading to 140 less total number of messages, or 46,620 total messages).

The number of multicast operations is constant for both time managed and unsynchronized DDM. Every LP is joined to the multicast group at the beginning. In our case, there are 20 multicast groups per LP that correspond to each of its own objects, and 7 other LPs, and consequently 140 = 20 x 7 multicast join operations are being issued.

Finally, since all multicast join and leave operations occur at the beginning of the simulation, there is no need to resend or retract any of the previously issued updates during the execution. Thus, the number of message resends and retracts equals zero for the synchronized DDM.

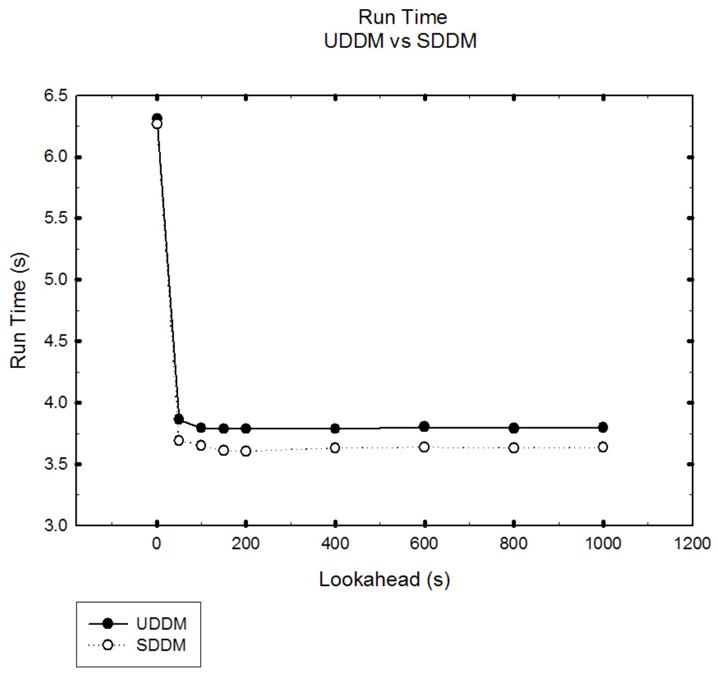


Figure 12 Run time for UDDM vs. SDDM.

Table 2 Run time confidence interval (95%) for SDDM

Lookahead	1	50	100	150	200	400	600	800	1000
95% Conf.	0.2164	0.0383	0.0282	0.0283	0.0278	0.0277	0.0257	0.0250	0.0232

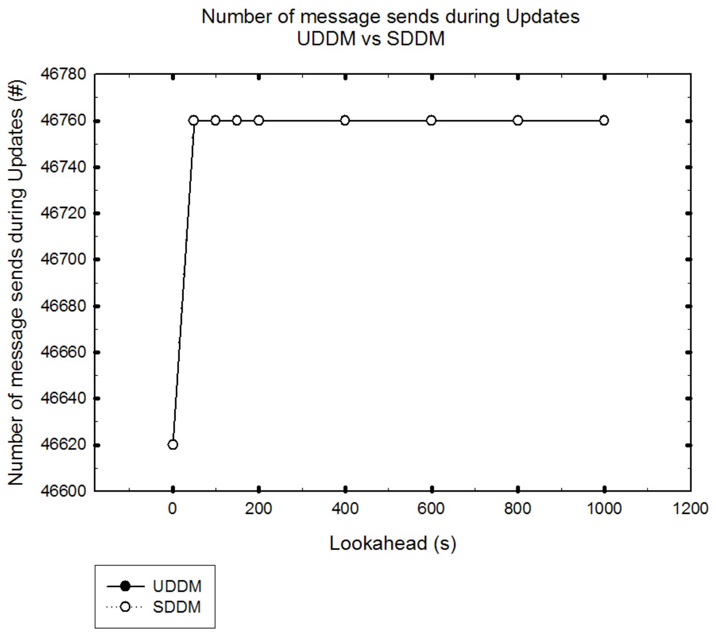


Figure 13 Number of message sends during Updates for UDDM vs. SDDM.

The next two figures show the overhead of DDM Update and Modify Region operations and where this overhead stands in comparison to the total simulation execution time. The total time for DDM operations is relatively constant over all lookahead values, and hence, a 50-fold increase in lookahead is the only reason that there is significant improvement in run time initially. It can be seen that the Update operations take less time than the Modify Region operations. The reason for this is that a region change involves computation and communication with other LPs, as well as adjusting the multicast groups by issuing multicast group join and leave operations. It can also be seen that DDM operations take no more than 50% of the total run time. The rest of the execution time is taken by waiting for time advances (i.e. for the LBTS computations to finish) and for receiving events and delivering updates.

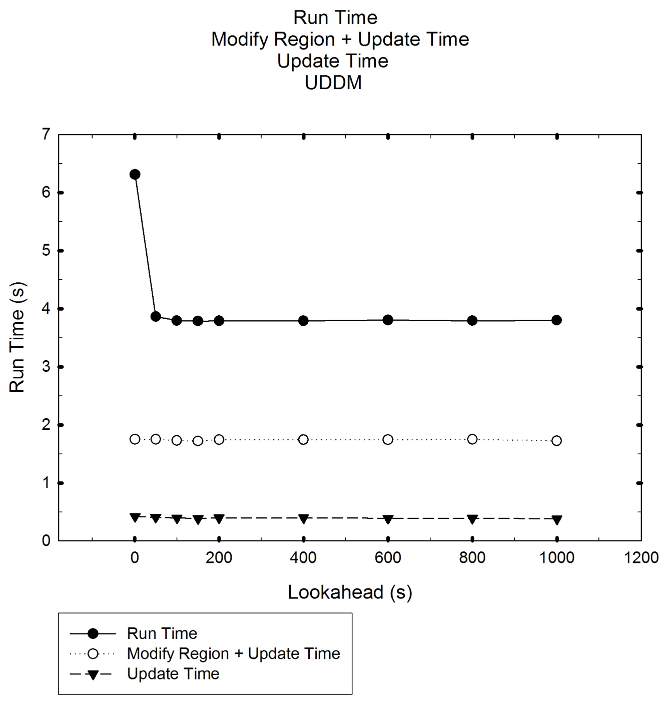


Figure 14 DDM operations overhead for UDDM.

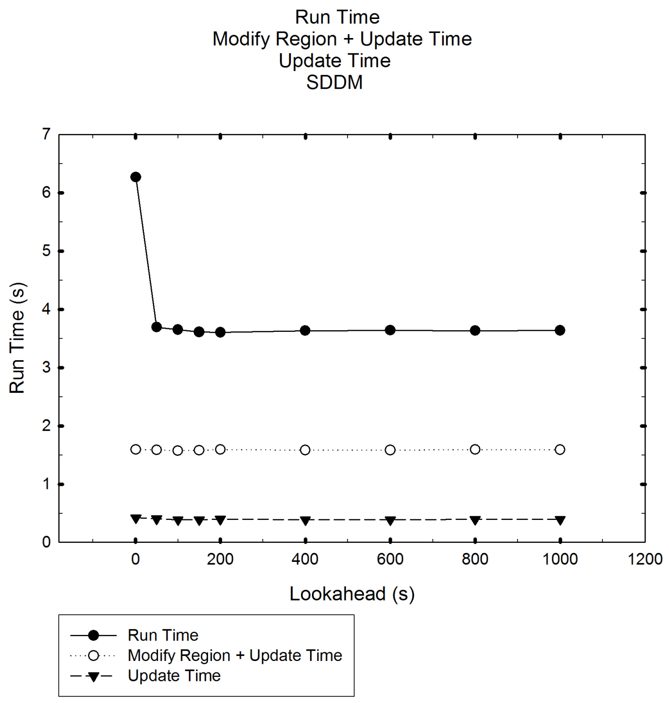


Figure 15 DDM operations overhead for SDDM.

4.4.2.2 Medium Region Size

Now we discuss what happens when regions are of medium size. Experiments for both FCS and the synthetic mobility patterns are presented in Figure 16 to Figure 27.

Medium region size parameters (also shown earlier in Table 1) are:

- *Update region diagonal distance = 1000 meters*
- *Visibility range = 10000 meters*

As can be seen (Figure 16 and Figure 17), synchronized DDM overheads become dominant after a certain lookahead value (i.e. for lookahead of 400 seconds). The number of message resends and retracts becomes greater with increasing lookahead and dominates all the other overheads starting at that specific lookahead value. This occurs when network send and receive buffers start to overflow from these messages, causing LPs to block. Greater lookahead values cause more resends / retracts which in turn cause more blockings in order to free up some buffers. An analysis of what affects this number of resends / retracts is presented later, in section 4.5. It can also be seen that the change in the number of messages sent during updates is much less profound than the change in the number of messages resent / retracted as lookahead increases (Figure 18 to Figure 21).

Finally, the number of multicast operations steadily decreases for time-managed DDM (Figure 22 and Figure 23), reducing the dominant overhead of message resends / retracts.

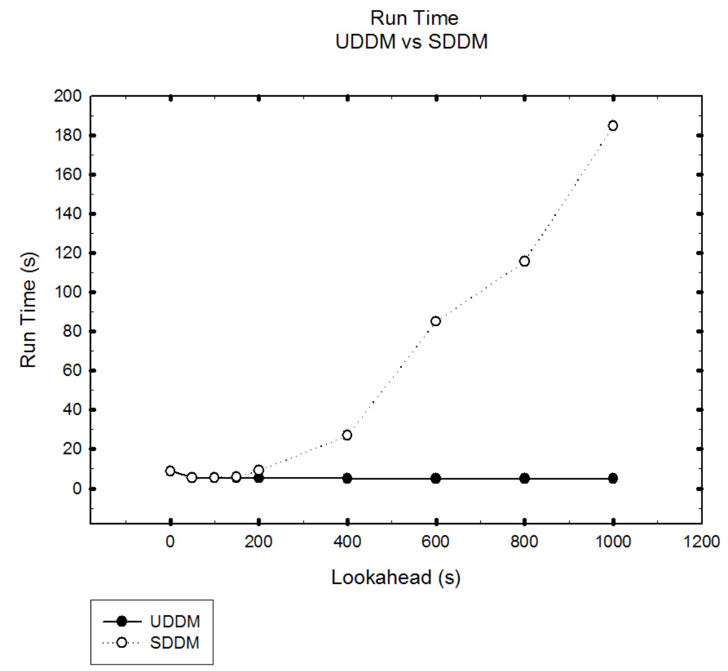


Figure 16 Run time for UDDM vs. SDDM - FCS.

Table 3 Run time confidence interval (95%) for SDDM - FCS

Look-ahead	1	50	100	150	200	400	600	800	1000
95% Conf.	0.2280	0.0367	0.0354	0.2831	0.9999	3.5119	8.5556	13.4968	16.7883

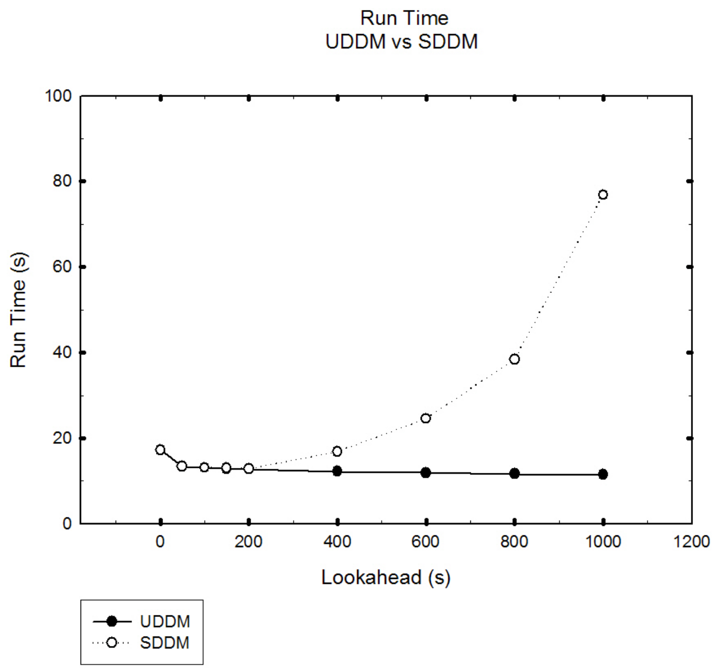


Figure 17 Run time for UDDM vs. SDDM - Synthetic.

Table 4 Run time confidence interval (95%) for SDDM - Synthetic

Lookahead	1	50	100	150	200	400	600	800	1000
95% Conf.	0.0495	0.0353	0.0262	0.0264	0.0224	1.3413	2.3265	6.7558	6.6581

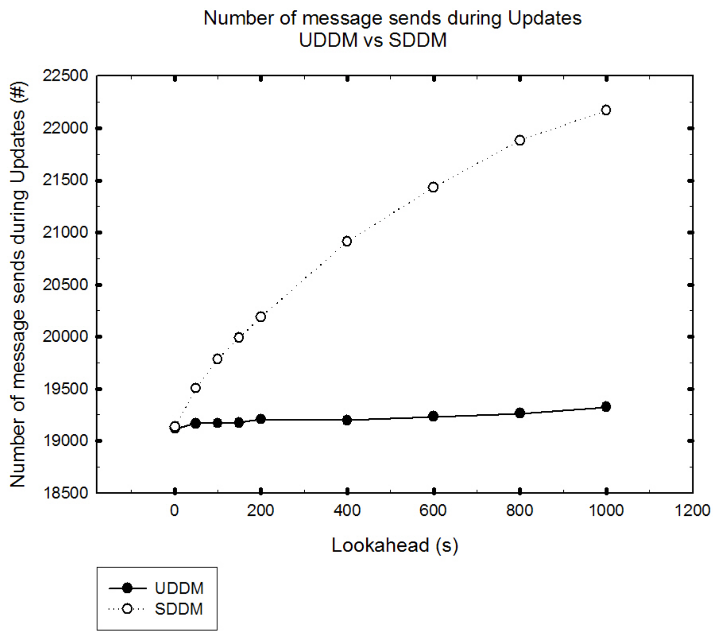


Figure 18 Number of message sends during Updates for UDDM vs. SDDM - FCS.

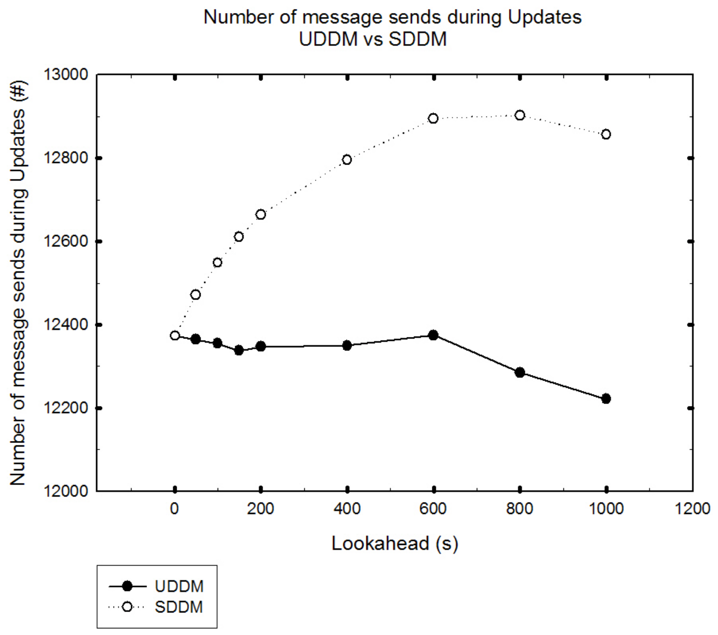


Figure 19 Number of message sends during Updates for UDDM vs. SDDM - Synthetic.

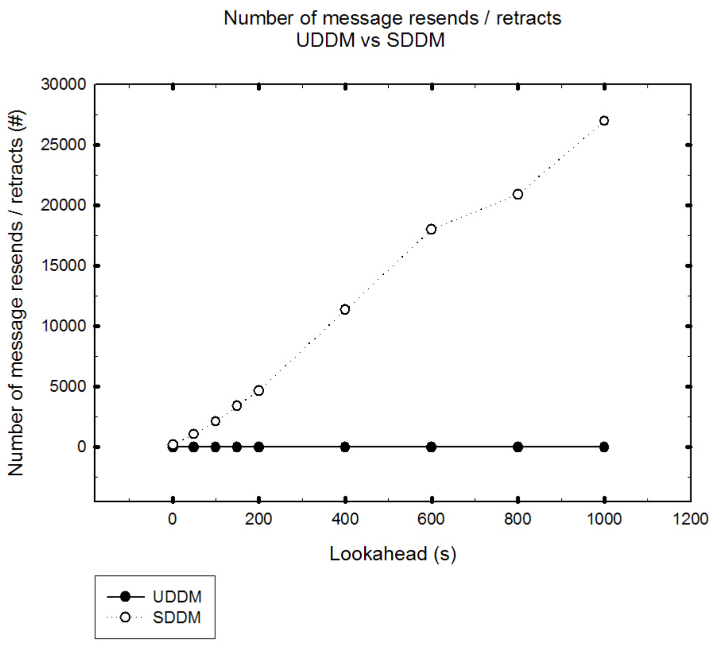


Figure 20 Number of message resends / retracts for UDDM vs. SDDM - FCS.

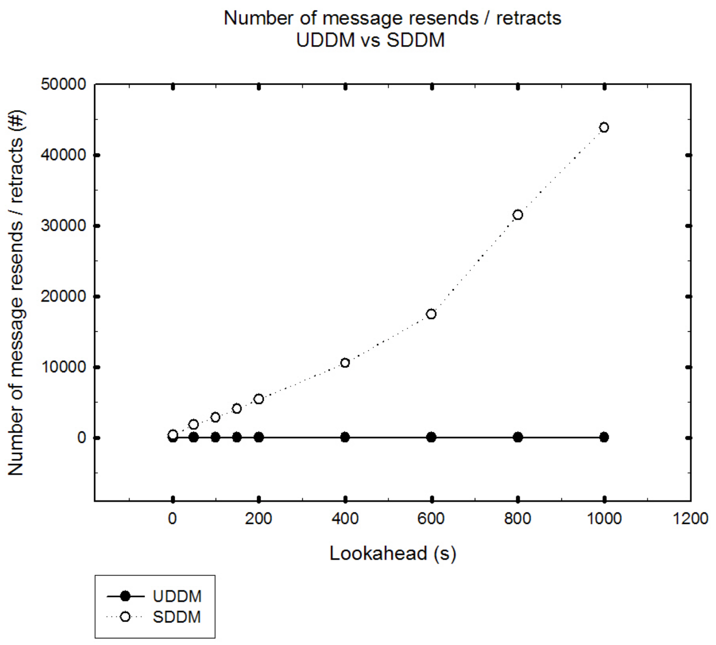


Figure 21 Number of message resends / retracts for UDDM vs. SDDM - Synthetic.

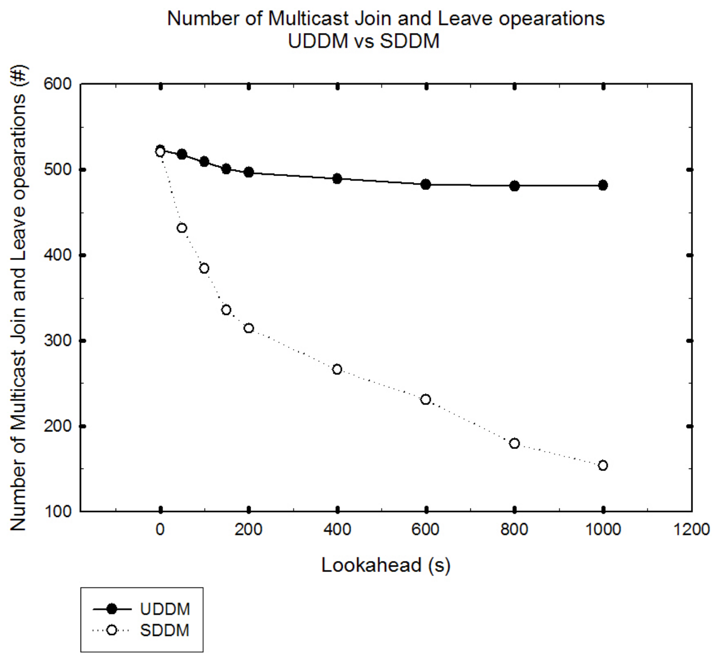


Figure 22 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM - FCS.

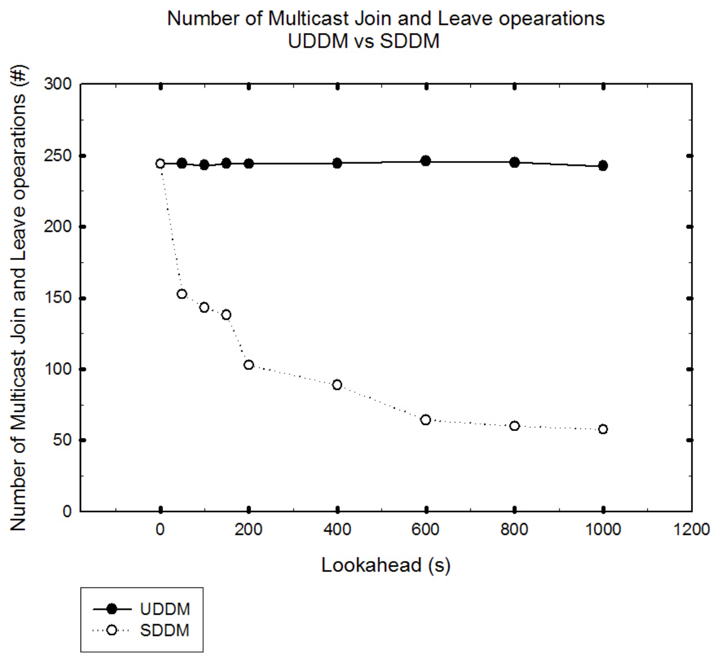


Figure 23 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM - Synthetic.

The overheads of each of the DDM operations are shown in Figure 24 and Figure 25. As expected, the Modify Region operation is the dominant DDM operation in the synchronized case because of the increasing number of message resends and retracts. At the same time, these extra messages increase the non-DDM time for time advances (i.e. LBTS computations) and for receiving events and delivering updates.

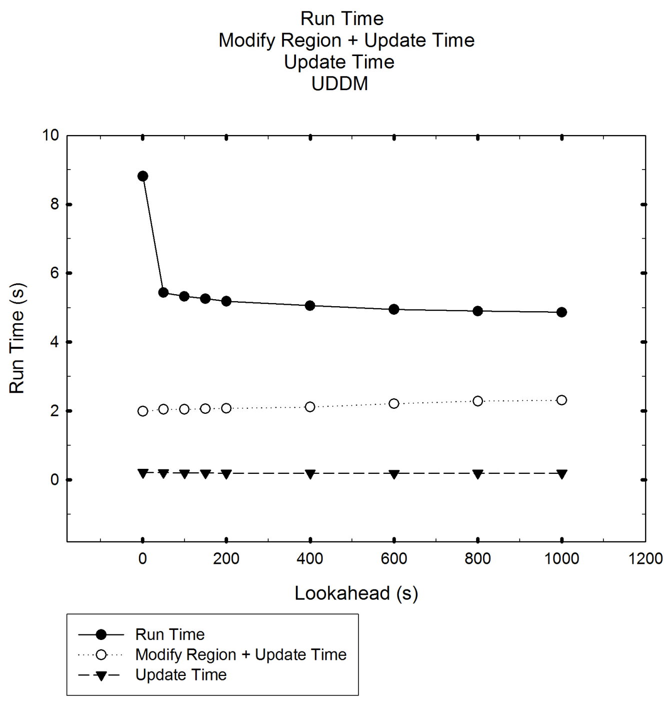


Figure 24 DDM operations overhead for UDDM - FCS.

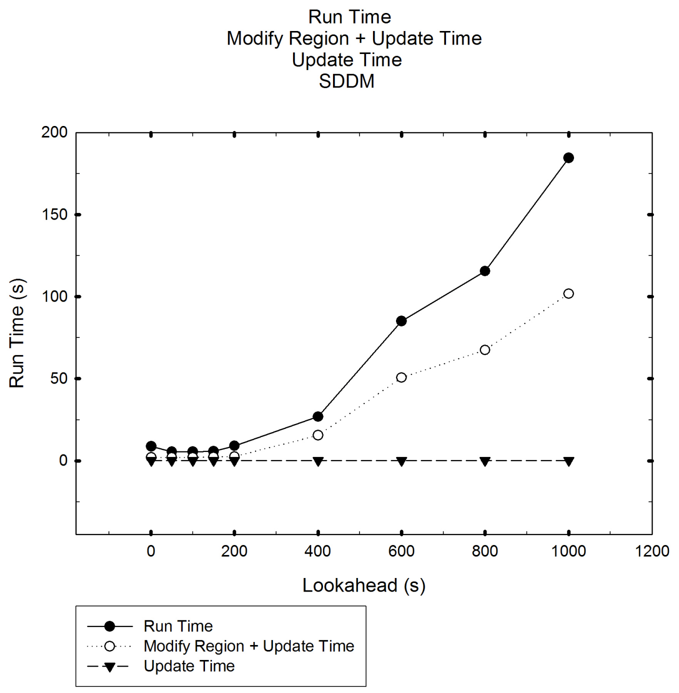


Figure 25 DDM operations overhead for SDDM - FCS.

Figure 26 and Figure 27 depict DDM operations overheads for the synthetic mobility pattern. It can be seen that they resemble the overheads for the FCS mobility patterns.

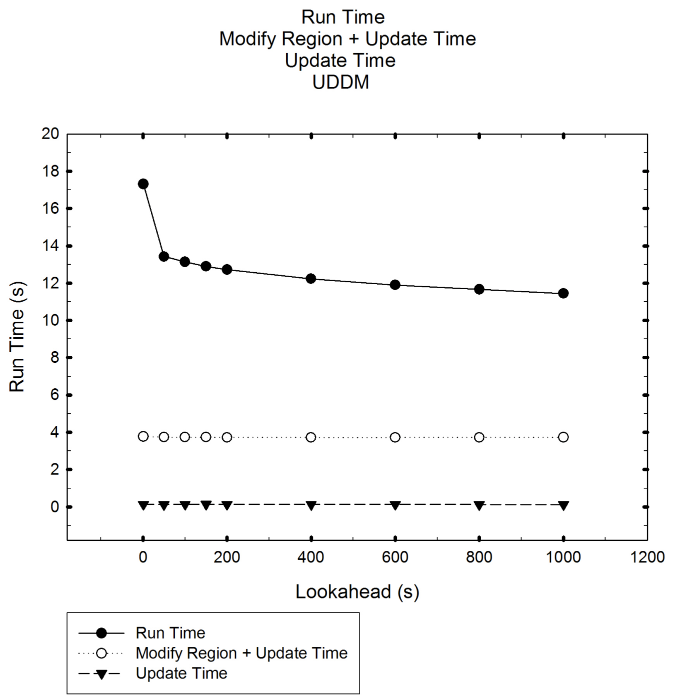


Figure 26 DDM operations overhead for UDDM - Synthetic.

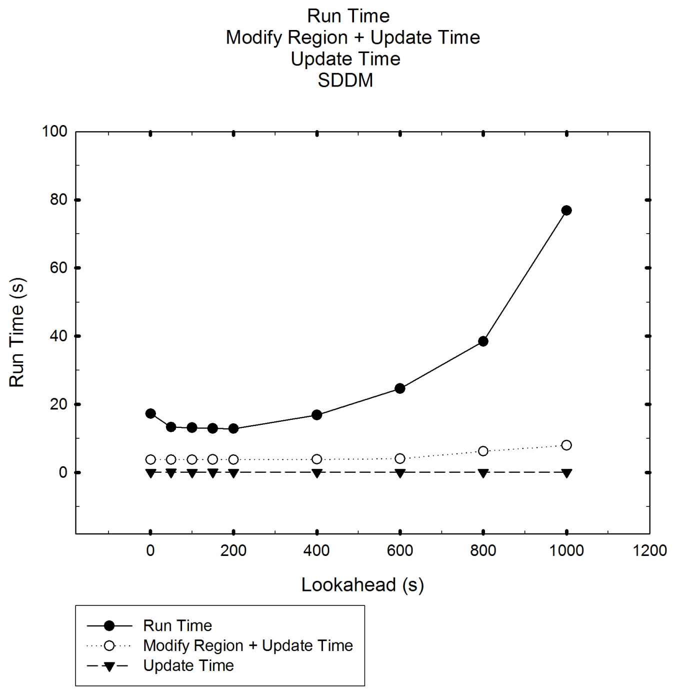


Figure 27 DDM operations overhead for SDDM - Synthetic.

4.4.3 Impact of Network Buffers on Performance

In this section we examine the effect of network buffers on performance. All the previous experiments from section 4.4.2 had the following buffer sizes:

- *recvbuff = 127680 bytes*
- *sendbuff = 127680 bytes*

More blocking and hence worse performance is expected by reducing the network buffers as illustrated in Figure 28. This is discussed next.

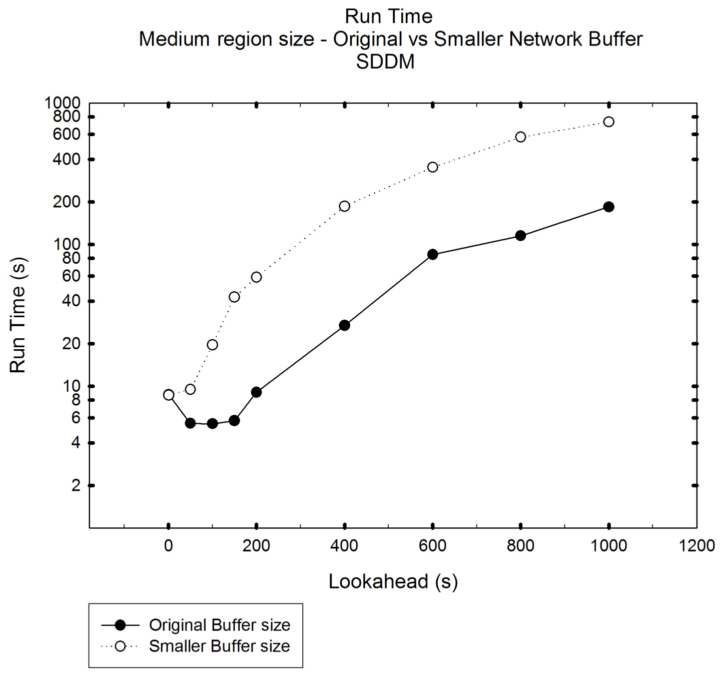


Figure 28 Run time when varying size of network buffers for SDDM.

4.4.3.1 Smaller Network Buffers

In this section we examine the effect of smaller network buffers for FCS mobility patterns. Buffers are reduced from 127680 bytes to these values:

- *recvbuff = 27680 bytes*
- *sendbuff = 27680 bytes*

As can be seen from the measurements (Figure 29 to Figure 34), performance starts to deteriorate much sooner than before, when network buffers were larger. This is because there is more blocking, since smaller buffers overflow sooner.

We can conclude that time-managed DDM performance is similar to unsynchronized DDM for small lookahead values. SDDM performance remains comparable to UDDM for somewhat larger lookahead values with larger network send and receive buffers for a particular mobility pattern. Performance is good so long as the number of network buffers are large enough to accommodate all message resends / retracts. To compensate for message overheads as lookahead is being increased, network buffers have to be increased at the same time to avoid or reduce the chance of blocking due to buffer overflows.

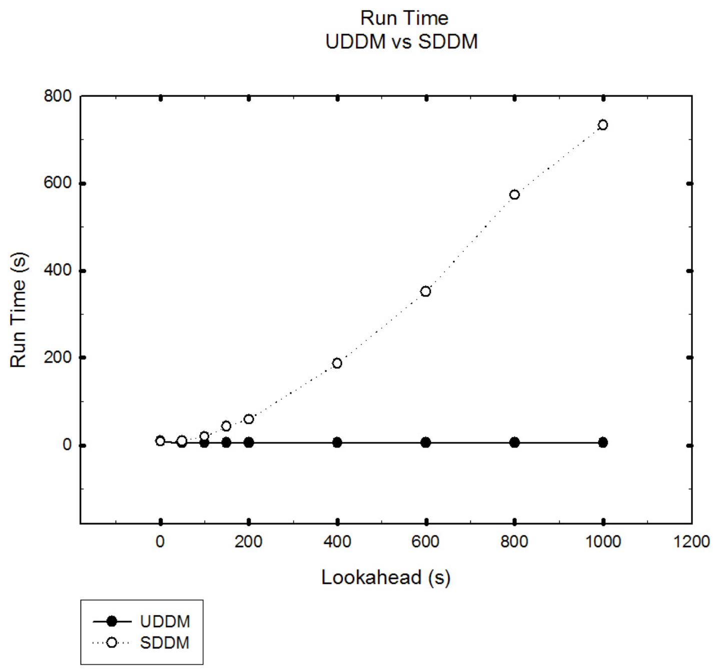


Figure 29 Run time for UDDM vs. SDDM.

Table 5 Run time confidence interval (95%) for SDDM

Look-ahead	1	50	100	150	200	400	600	800	1000
95% Conf.	0.0452	1.5242	3.4358	6.4778	7.6771	30.9987	57.4148	75.8584	106.4069

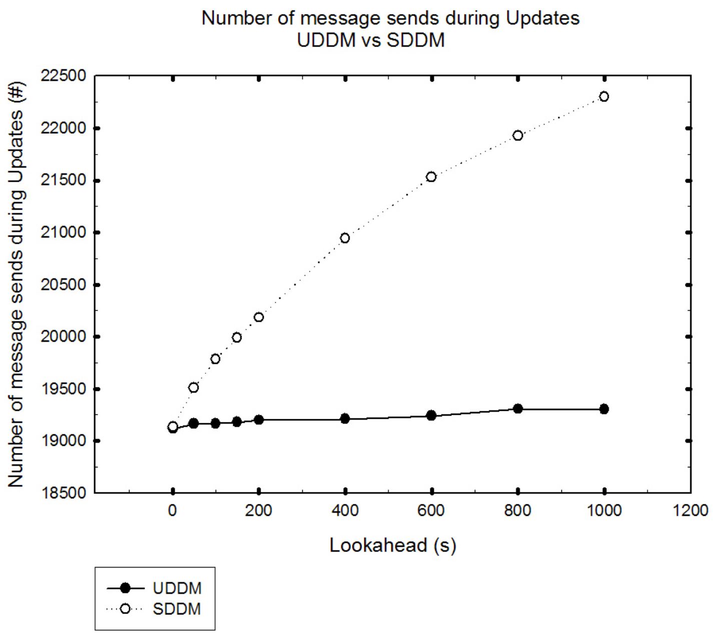


Figure 30 Number of message sends during Updates for UDDM vs. SDDM.

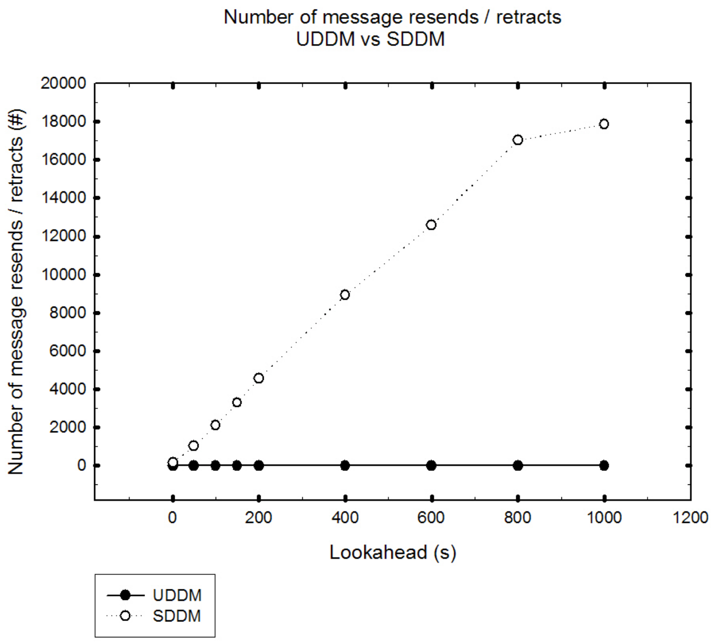


Figure 31 Number of message resends / retracts for UDDM vs. SDDM.

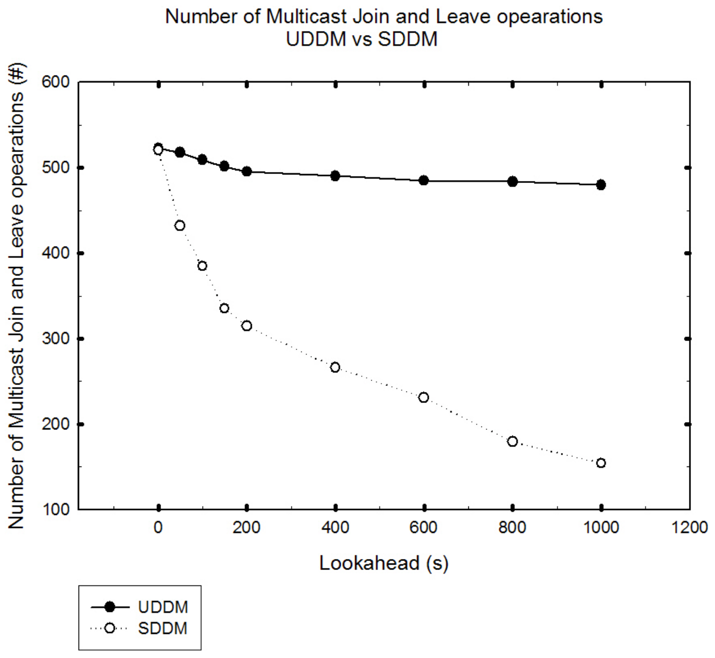


Figure 32 Number of multicast operations (Joins/Leaves) for UDDM vs. SDDM.

The effects of smaller network buffers on each of the DDM operations can be seen in Figure 33 and Figure 34. Although the run time overhead is significantly larger, the DDM operations affect performance similarly over lookahead as with the original buffer sizes.

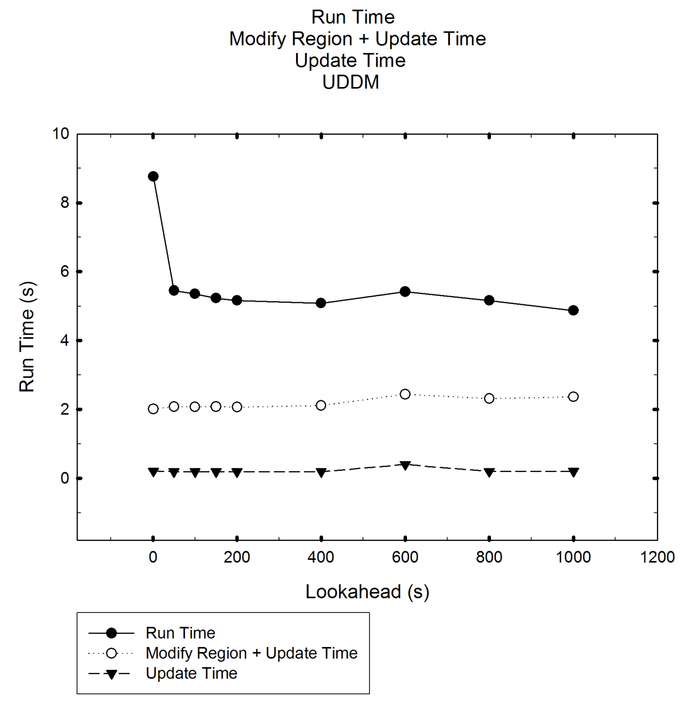


Figure 33 DDM operations overhead for UDDM.

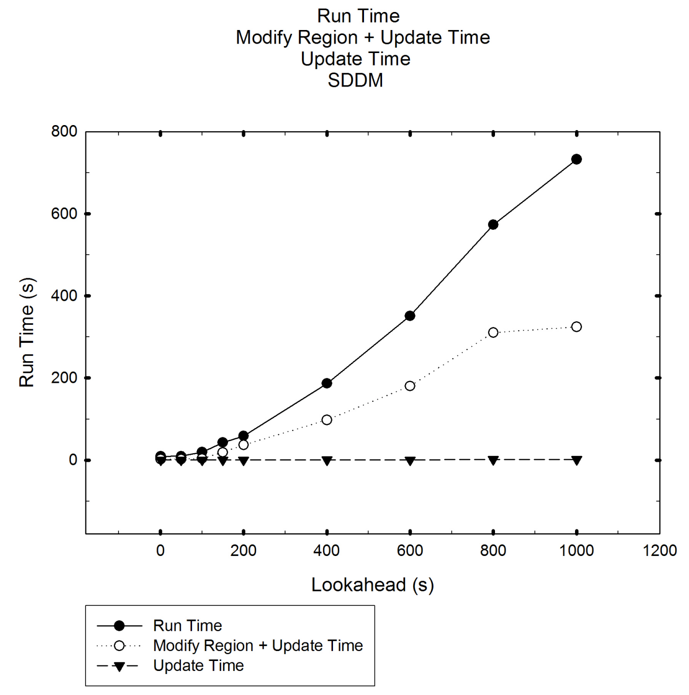


Figure 34 DDM operations overhead for SDDM.

4.5 Analysis

The previous section showed that message resends and retracts are the major factors that affect the performance of synchronized DDM. In this section, we will give an upper bound estimate on the number of message resends / retracts, and show how this number grows in terms of other application specific and execution parameters such as the frequency of region changes, frequency of updates and lookahead.

We will use the following parameters in our analysis to compute upper bounds of resend / retract messages:

$\boldsymbol{L, f_L = 1 / L}$ – lookahead, and corresponding frequency defined as 1 / L

$\boldsymbol{f_{MR}}$ – frequency of Modify Region operations

$\boldsymbol{f_U}$ – frequency of Update operations

$\Delta \boldsymbol{T}$ – simulation execution duration

$\boldsymbol{\#U_{RR}}$ - number of resend/retract update messages (upper bound)

We can express the total number of resends / retracts in terms of the effective frequency of update resends / retracts ($\boldsymbol{f_{URReffective}}$) that we will compute next:

$$\boldsymbol{\#U_{RR} = \Delta T * f_{URReffective}}$$

The total number of updates resent / retracted can be computed as the number of updates resent / retracted for each modify region operation ($\#U_{RR\_\forall MR}$) during the execution, that is:

$$\#U_{RR} = \#U_{RR\_\forall MR} * \#MR_{in\_\Delta T}$$

where:

$\#MR_{in\_\Delta T}$ is the number of modify region operations during the execution duration ΔT.

The upper number of resends / retracts per modify region is simply the maximum number of updates that can fit into a lookahead interval, that is:

$$\#U_{RR\_\forall MR} = f_U * L.$$

The number of modify regions during the execution is:

$$\# MR_{in\_\Delta T} = f_{MR} * \Delta T.$$

Thus, we can rewrite the total number of updates resent / retracted and corresponding effective frequency as:

$$\#U_{RR} = (f_U * L) * (f_{MR} * \Delta T)$$

$$\#U_{RR} = \Delta T * [\, f_U * (\, f_{MR} / f_L \,)] = \Delta T * f_{URReffective}$$

$$f_{URReffective} = f_U * (\, f_{MR} / f_L \,)$$

In our implementation, effective frequency of updates resent / retracted grows more rapidly:

$$f_{URReffective} = f_U * (f_{MR} / f_L)^2$$

Although less efficient than the upper bound we computed previously, we are able to see the effects of modify region frequency (***f_{MR}***) and lookahead (***L*** or ***f_L***) sooner with our implementation, since effective resent/retract frequency grows quadratically in respect to the ***f_{MR} / f_L*** ratio.

The explanation for this behavior follows. Late subscription region change can cause a sequence of group membership changes (i.e. join and leave operations) in the lookahead interval, starting at the LP's current logical time. As described previously, it suffices to retract and resend updates in the lookahead interval (formula $\#U_{RR\_\forall MR} = f_U * L$ above) all at once. In our implementation, however, we process these group membership changes separately, so they are causing more updates to be resent / retracted. This has the same effect, since after all retractions and message resends, exactly the same updates will be delivered to the LP. On the other hand it results in more messages.

In the worst case, when a late subscription region change has a time stamp very close to the current time, we have to resend / retract all updates in this lookahead interval first. This change will cause the group membership to change at a time stamp of the previously issued update region change, which is 1 / f_{MR} away in the future, which in turn, causes resend / retracts from that logical time to the end of lookahead interval. Similarly, this process continues at every previous update region change, that is 1 / f_{MR} away in the future. The number of updates that have to be resent / retracted decreases at each step by the number of updates found in 1 / f_{MR}, that is by f_U / f_{MR}. The total number of updates resent / retracted can be written as an arithmetic sum:

$$\#U_{RR\_\forall MR} = f_U / f_{MR} * (1 + 2 + \ldots + k)$$

$$= f_U / f_{MR} * (k * (k+1) / 2)$$

$$\approx f_U / f_{MR} * k^2$$

where ***k*** is the number of modify regions in a lookahead interval:

$$k = f_{MR} / f_L$$

Thus, the upper number of resends / retracts per modify region is:

$$\#U_{RR\_\forall MR} \approx f_U / f_{MR} * (f_{MR} / f_L)^2$$

$$\approx f_U * f_{MR} * / f_L^2$$

Finally, total number of resends / retracts and corresponding effective frequency during the execution is:

$$\#U_{RR} = \#U_{RR\_\forall MR} * \#MR_{in\_\Delta T}$$

$$\approx f_U * f_{MR} * / f_L^2 * f_{MR} * \Delta T$$

$$\#U_{RR} \approx \Delta T * [f_U * (f_{MR} / f_L)^2] = \Delta T * f_{URReffective}$$

$$f_{URReffective} \approx f_U * (f_{MR} / f_L)^2$$

Observations:

1) $f_{MR} << f_L$ implies $f_{MR} / f_L \to 0$

That is, there is an insignificant number of message resend/retracts if lookahead frequency dominates, or in other words if lookahead is much larger than the average distance between region changes.

2) $f_{MR} = f_L$ implies $f_{MR} / f_L = 1$

That is, the number of message resends / retracts equals the number of updates originally issued (with f_U frequency) when lookahead equals average distance between modify region operations.

3) $f_{MR} >> f_L$ implies $f_{MR} / f_L \to \infty$

That is, a significant number of resends / retracts lookahead is much smaller than the average distance between modify region operations.

4.6 Performance Summary

The experiments and analysis in this chapter show that synchronized DDM performance depends on the lookahead, the application's mobility patterns and the network hardware. Time-managed DDM services can be made as efficient as unsynchronized DDM services in the workstation cluster environment as long as we can compensate for increases in lookahead with larger network buffers.

Time-managed DDM services are as efficient as unsynchronized DDM services when the mobility patterns cause infrequent group membership changes. This can be seen in section 4.4.2.1 where region sizes are large, so that they do not cause frequent group membership changes.

However, synchronized and unsynchronized DDM performance can be quite different with general mobility patterns. This is the case for both FCS and synthetic applications when network buffers are not large enough to be able to handle all resend / retract messages without blocking. Section 4.4.2.2 shows that for the medium region size and typical size of the network buffers, SDDM performance is similar to UDDM up to a certain lookahead value. This lookahead value is greater with larger network send and receive buffers for a particular mobility pattern. Section 4.4.3 confirms this by decreasing network buffer size when performance overhead of time-managed DDM becomes significant sooner than for typical size of network buffers from section 4.4.2.2.

Furthermore, synchronized DDM overheads become dominant after a specific lookahead value which depends on the size of network buffers, as was just explained. The number of message resends / retracts becomes greater with lookahead (and mobility patterns according to formula for frequency of message resends / retracts from section 4.5: $f_{URReffective} = f_U * (f_{MR} / f_L)$), and hence network send and receive buffers get filled up sooner, causing more blockings.

Solely having larger lookaheads for time-managed DDM does not necessarily translate into better performance. On one hand, increasing lookahead for unsynchronized DDM improves performance, since more concurrent events can be executed until the next LBTS (lower bound time stamp) value has to be computed. However, the number of message resends / retracts grows with the lookahead and the application's mobility patterns for synchronized DDM according to the analysis from previous section. To compensate for this message overhead, network buffers have to be increased at the same time to avoid or reduce the chance of blocking due to buffer overflows.

Thus, we conclude that time-managed DDM services are as efficient as unsynchronized DDM services in the workstation cluster environment as long as we can compensate for increases in lookahead with larger network buffers. How much increase in network buffers is needed depends on the application's mobility patterns. When mobility patterns cause infrequent group membership changes, there is no need for a significant buffer increase. On the other hand, frequent group membership changes entail buffer increases to accommodate for the increase in message resends / retracts.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Summary

Data distribution management (DDM) is a mechanism to interconnect data producers and data consumers in a distributed application. Data must be routed between producers and consumers. For each message produced, DDM determines the set of consumers interested in receiving the message and delivers it to those consumers.

An efficient DDM implementation must be realized. Broadcast is one example of a simple, but inefficient DDM implementation. Two issues must be addressed when realizing DDM. The amount of unnecessary messages flowing between data producers and data consumers makes the network bandwidth requirements excessively large. Second, unnecessary messages cause processing delays because data consumers must filter unwanted messages. Hence, the DDM system should minimize the total number of messages that must be sent to link producers and consumers in order to realize scalable distributed applications.

We are particularly interested in DDM techniques for parallel and distributed discrete event simulations. Thus far, researchers have treated synchronization of events (i.e. time management) and DDM independent of each other. Time managed services employ a logical time abstraction giving one explicit control over the order in which these services are observed by the participating logical processes. This is the case in analytical simulations which are used to analyze systems by mimicking the causal relationships (i.e. before and after relationships) of a physical system precisely or as closely as possible. They are typically used to obtain accurate statistics about the behavior of the physical system being modeled (e.g. modeling of a telecommunication network). On the other hand, non-time managed services are based on wallclock time which does not provide such ordering guarantees. This may be acceptable to some simulations, such as real-time training simulations where before and after relationship may not always be perceptible by human participant, so casual relationships can sometimes be relaxed.

This research focuses on how to realize time managed DDM mechanisms. The main reason for time-managed DDM is to ensure that changes in the routing of messages from producers to consumers occur in a correct sequence (e.g. before and after relationships in analytical simulations). Also time managed DDM avoids non-determinism in the federation execution, which may result in non-repeatable executions. Directly applying training simulation DDM mechanisms based on wallclock time semantics to logical time simulations will lead to errors. In this research we show how to efficiently synchronize DDM.

An "optimistic" approach to time managed DDM is proposed where one allows events to be processed out of time stamp order, and an error detection and recovery procedure is used rather than strictly avoiding errors. Furthermore, it is tailored to the semantics of the DDM services, thereby avoiding the use of general rollback mechanisms, which are more complex and require large runtime overheads.

At the core of our optimistic approach is avoiding missed messages, which may occur due to late subscription region changes. We utilize a message log which records messages as they are sent. When a new DDM event indicates that a previously generated message should have been sent to an LP, but in fact was not, the message is retrieved from the log and sent. Thus, missed messages are avoided.

We also address the problem of matching producers and consumers of data, while keeping the number of messages that are generated at a minimum. A hybrid approach is developed that deals with this issue for time managed as well as non-time managed DDM systems while maintaining low computational and message overheads. To achieve this, a multicast group is defined for each update region, eliminating the duplicate and extra message problem of the grid scheme. Furthermore, grid partitioning is used to speed up matching of update and subscription regions by considering only a subset of all the regions that overlap the grid cells recently covered. This improves the scalability of the pure update-region based approach.

We have developed a fully distributed implementation of the algorithm within the framework of the Georgia Tech Federated Simulation Development Kit (FDK) software. This implementation is used to evaluate the approach. A performance evaluation of the synchronized DDM mechanism has been completed in a loosely coupled distributed system consisting of a network of workstations connected over a LAN. We compare time-managed versus unsynchronized DDM for two applications that exercise different mobility patterns: Future Combat Systems and synthetic benchmark application.

The experiments and analysis show the trade-offs on performance along the following three dimensions:

- the simulation's model (e.g. lookahead)
- application's mobility patterns
- network hardware and software (e.g. size of network buffers)

It was shown that larger lookaheads are not always sufficient for better performance of time-managed DDM. On one hand, larger lookahead helps in increasing the number of events that can be executed in parallel on different logical processes, and hence, it helps achieve better performance for unsynchronized DDM. However, depending on the mobility patterns, it can also cause significant growth in the number of message resends / retracts for synchronized DDM. This message overhead needs to be compensated by larger network buffers that will be able to accommodate all the message resend / retracts, and hence, reduce the chance of blocking due to buffer overflows.

Under certain mobility patterns, time-managed DDM is as efficient as unsynchronized DDM. This is the case when the mobility patterns cause infrequent group membership changes. For example, large regions do not cause frequent group membership changes. Even if group membership changes are more frequent, there is not much difference in performance when network buffers are large enough to be able to handle all resend / retract messages without blocking.

On the other hand, there are also mobility patterns where time-managed DDM overheads become significant. Synchronized DDM overheads become dominant after a specific lookahead value which depends on the size of network buffers. The number of message resends / retracts becomes greater with lookahead, and hence network send and receive buffers get filled up sooner, causing more blockings. It was shown that the number of message resends / retracts grows linearly or even quadratically in respect to f_{MR} / f_L ratio (i.e. modify region frequency / lookahead frequency ratio), depending on the implementation. To overcome this problem, increase in lookahead has to be compensated with larger network buffers.

Thus, we conclude that time-managed DDM services are as efficient as unsynchronized DDM services in the workstation cluster environment as long as we can compensate increases in lookahead with larger network buffers. Network buffers need to be increased depending on the application's mobility patterns. While infrequent group membership changes do not require a significant buffer increase, frequent group membership changes entail appropriate buffer increases to accommodate for the rise in message resends / retracts.

5.2 Future Work

There are several open issues with respect to the time managed DDM approach, some of which are listed in the following sections.

5.2.1 Precise Filtering

Precise filtering is one of the paths to explore in future synchronized DDM implementations. The preciseness of our hybrid approach which performs matching of update and subscription regions depends on the chosen cell size relative to the region sizes. In fact, the hybrid approach also yields a perfect matching when regions always fully overlap grid cells. In general, the hybrid approach is less efficient but more precise as the cell size decreases since there are more cells to consider, and vice versa.

Our approach can easily be extended to produce a perfect matching, by defining two additional cell and region counters. One pair of cell and region counters can be used for cells that are fully overlapped, while the other pair of counters can be used for partially overlapped cells. When there is at least one cell fully overlapped by both update and subscription regions at every logical time, the extended hybrid algorithm mimics the execution of the original hybrid algorithm. Only counters for the fully overlapped cells are used to perform the matching and generate multicast group join and leave operations, while the other set of counters are only being updated. The second set of counters assigned for partially overlapped cells is used to perform matching when the original region counter's value is zero at some logical time. In these situations the extended hybrid algorithm checks if there is at least one non-zero cell counter for partially overlapped cells. If that is the case, further comparison of an update region and subscription region's partially overlapping cells of interest is necessary in order to determine possible distribution list changes (i.e. new Add or Delete operations from the Distribution List layer) and to make required multicast group membership changes.

The extended hybrid approach to matching is not expected to add significant overhead to the existing hybrid algorithm. Most of the time it is expected to perform just the same as the original counterpart, since partial cell and region counters will need to be considered only when regions start to overlap as objects move into each other's visibility range, or as they leave each other's visibility range.

5.2.2 Utilizing Additional Multicast Groups

Our synchronized DDM approach assigns a multicast group to each update region, and hence the total number of multicast groups required depends only on the number of update regions for all the LPs in the simulation. However, this can cause extra messages since one multicast group cannot capture all the distribution list changes from an LP's current logical time into the future. It was explained that an LP may be joined to a multicast group even if that LP is not subscribed (i.e. deleted) at some logical time intervals from its current time into the future. This could result in extra update messages that have to be discarded at the destination LPs.

Our approach can be enhanced with more multicast groups per update regions' timeline, by dedicating a multicast group per a distinct portion of the timeline. For example, with $K > 1$ multicast groups available per update region, we can divide a timeline segment $[T + L, \infty)$ where T is LP's current time, into K smaller non-overlapping segments, and assign a multicast group for each of the segments. This will reduce or even completely eliminate extra messages that may occur in situations described above. If there were enough multicast groups available to be assigned for every distribution list's Add / Delete operation, extra messages would have been completely eliminated. Unfortunately, one must take into account the overheads and delays when using the multicast group communication. There is a tradeoff between how many and how frequently multicast groups are used and the number of extra messages. Using more groups per update region's time line will be beneficial only up to a certain point. This and other tradeoffs when utilizing additional network multicast resources remain to be explored in the future.

5.2.3 Automatic Self-adjusting DDM Mechanism for Improved Performance

Time managed DDM algorithms which automatically identify and adjust the optimal lookahead value are important for keeping the number of message resends and retracts from being too high. As we have seen from the experiments and the analysis, the number of such messages grows linearly or quadratically with lookahead, depending on the implementation. On the other hand, larger lookaheads allow more concurrency and faster time advances during the simulation executions. Thus, for a given model's lookahead, DDM mechanism could observe the simulation speed and the number of message resends and retracts over a time period, and then would make adjustments to the lookahead as necessary. Lookahead can always be decreased or increased safely throughout the simulation execution as long as it does not exceed the maximum amount provided for the model.

APPENDIX A

FULL IMPLEMENTATION OF THE ALGORITHM

Modify subscription region(subscription region SR, federate F, logical time T){

```
1. Phase 1: Update Cell Layer counter
      • Determine all new/old cells C for SR    // from SR(T) and previous instance in
        time of SR
      • Determine $T_{end}$; $T_{start}$= T
      • ∀ new cells issue the following **CL::subscribe( C, F, $T_{start}$, $T_{end}$)** operation:
          **CL::subscribe( C, F, $T_{start}$, $T_{end}$){**
            // >>> If no entry at $T_{start}$ create one
            find closest counter $C_{sbsc}$[F]: strength@t such that t <= $T_{start}$;
            if( t < $T_{start}$) create new counter $C_{sbsc}$[F]: strength@$T_{start}$ = $C_{sbsc}$[F]:
        strength@t;

            // >>> If no entry at $T_{end}$ create one
            find closest counter $C_{sbsc}$[F]: strength@t such that t <= $T_{end}$;
        if( $t < T_{end}$) create new counter $C_{sbsc}[F]$: strength@$T_{end}$ = $C_{sbsc}[F]$: strength@t;

        $\forall$ counter in $t \in [T_{start}, T_{end})$:
                ++ $C_{sbsc}[F]$: strength@t ;

    }
```

- $\forall$ *old cells issue the following* ***CL::unsubscribe(C, F, T_{start}, T_{end})*** *operation:*

```
    CL::unsubscribe( C, F, $T_{start}$, $T_{end}$){
        // >>> If no entry at $T_{start}$ create one
        find closest counter $C_{sbsc}[F]$: strength@t such that $t <= T_{start}$;
        if( $t < T_{start}$) create new counter $C_{sbsc}[F]$: strength@$T_{start}$ = $C_{sbsc}[F]$: strength@t;

        $\forall$ counter in $t \in [T_{start}, T_{end})$:
                -- $C_{sbsc}[F]$: strength@t ;

    }
```

2. *Phase 2: Update Region Layer counter*
   - $\forall$ *new cells C*

```
$\forall$ UR overlapping cell C in $t \in [T_{start}, T_{end})$
    $\forall$ overlapping time intervals $[T_{so}, T_{eo}) \in [T_{start}, T_{end})$:
 // >>> If no entry at $T_{start}$ create one
        if( $T_{so} = T_{start}$){
            find closest counter $U_{cumulative}$[UR][F]: strength@t such that t
        <= $T_{start}$;
            if( t < $T_{start}$) create new counter $U_{cumulative}$[UR][F]:
   strength@$T_{start}$ = $U_{cumulative}$[UR][F]: strength@t;

        }

        $\forall$ counter $U_{cumulative}$[UR][F]: strength@t where $t \in [T_{so}, T_{eo})$:
            ++ $U_{cumulative}$[UR][F]: strength@t ;
```

- $\forall$ old cells C

```
$\forall$ UR overlapping cell C in $t \in [T_{start}, T_{end})$
    $\forall$ overlapping time intervals $[T_{so}, T_{eo}) \in [T_{start}, T_{end})$:
 // >>> If no entry at $T_{start}$ create one
        if( $T_{so} = T_{start}$){
            find closest counter $U_{cumulative}$[UR][F]: strength@t such that t
        <= $T_{start}$;
```

```
            if( $t < T_{start}$) create new counter $U_{cumulative}[UR][F]: strength@T_{start} = U_{cumulative}[UR][F]: strength@t$;

        }

        $\forall$ counter $U_{cumulative}[UR][F]: strength@t$ where $t \in [T_{so}, T_{eo})$:
            -- $U_{cumulative}[UR][F]: strength@t$ ;

3.  Phase 3: Issue **Add/Delete** operations according to change in Region Layer counter
    •   $\forall$ UR where $U_{cumulative}[UR][F]: strength@t$ changes its value from 0 to greater than 0 in $t \in [T_{start}, T_{end})$:
            issue **Add**( F, UR, t)
    •   $\forall$ UR where $U_{cumulative}[UR][F]: strength@t$ changes its value from greater than 0 to 0 in $t \in [T_{start}, T_{end})$:
            issue **Delete**( F, UR, t)
}
```

Modify update region(update region UR, logical time T){

```
1.  Phase 1: Update Region Layer counter
    •   Determine all new/old cells C for UR
    •   Determine $T_{end}$; $T_{start}$= T
• ∀F in $t \in [T_{start}, T_{end})$:
    • ∀new cells C
        // >>> If no entry at $T_{start}$ create one
        find closest counter $U_{cumulative}$[UR][F]: strength@t such that t <= $T_{start}$;
        if( t < $T_{start}$) create new counter $U_{cumulative}$[UR][F]: strength@$T_{start}$ =
    $U_{cumulative}$[UR][F]: strength@t;

        // >>> If no entry at $T_{end}$ create one
        find closest counter $U_{cumulative}$[UR][F]: strength@t such that t <= $T_{end}$;
        if( t < $T_{end}$) create new counter $U_{cumulative}$[UR][F]: strength@$T_{end}$ =
    $U_{cumulative}$[UR][F]: strength@t;

        issue the following merge( merge type: +, C, F, $T_{start}$, $T_{end}$) operation that
        will update $U_{cumulative}$[UR][F]: strength@t in the positive (+) direction:
        merge( merge type: +/-, C, F, $T_{start}$, $T_{end}$){
          $U_{cumulative}$_next_time = $T_{start}$; // entry at $T_{start}$ exists
          $C_{sbsc}$_next_time = 1) time of the closest $C_{sbsc}$[F]: strength@t entry such
        that t >= $T_{start}$, or 2) $\infty$ if such an entry does not exist
    time = min($U_{cumulative}$_next_time, $C_{sbsc}$_ next_time);
while( time < $T_{end}$){
  if( time = $U_{cumulative}$_next_time){
            // $U_{cumulative}$_previous_strength is strength before merge operation
            $U_{cumulative}$_previous_strength = $U_{cumulative}$[UR][F]: strength@time;
                $U_{cumulative}$_next_time = 1) time of the next $U_{cumulative}$[UR][F]:
            strength@t entry, or 2) $\infty$ if such an entry does not exist
                if( $C_{sbsc}$_next_time = time)
             $C_{sbsc}$_next_time = 1) time of the next $C_{sbsc}$[F]: strength@t entry, or 2)
            $\infty$ if such an entry does not exist
              }
  else{   // time = $C_{sbsc}$_next_time < $U_{cumulative}$_next_time
// >>> create $U_{cumulative}$[UR][F] entry @ time = $U_{cumulative}$_previous_strength;
                $U_{cumulative}$[UR][F]: strength@ time = $U_{cumulative}$_previous_strength;
                $C_{sbsc}$_next_time = time of the next $C_{sbsc}$[F]: strength@t entry;
              }
            $U_{cumulative}$[UR][F]: strength@time    // this is now defined in both cases
                +/- = $C_{sbsc}$[F]: strength@ time;
    time = min($U_{cumulative}$_next_time, $C_{sbsc}$_ next_time);
           }
}
```

- $\forall$ old cells C

  // >>> If no entry at T_{start} create one

  find closest counter $U_{cumulative}$[UR][F]: strength@t such that t <= T_{start};

```
        if( $t < T_{start}$) create new counter $U_{cumulative}$[UR][F]: strength@$T_{start}$ = $U_{cumulative}$[UR][F]: strength@t;
        // >>> If no entry at $T_{end}$ create one
        find closest counter $U_{cumulative}$[UR][F]: strength@t such that $t <= T_{end}$;
        if( $t < T_{end}$) create new counter $U_{cumulative}$[UR][F]: strength@$T_{end}$ = $U_{cumulative}$[UR][F]: strength@t;

        // issue merge operation that will update $U_{cumulative}$[UR][F]: strength@t in the negative (-) direction
        issue merge( merge type: -, C, F, $T_{start}$, $T_{end}$) operation

2. Phase 2: Issue Add/Delete operations according to change in Region Layer counter
    • ∀ F where $U_{cumulative}$[UR][F]: strength@t changes its value from 0 to greater than 0 in $t \in [T_{start}, T_{end})$:
        issue Add( F, UR, t)
    • ∀ F where $U_{cumulative}$[UR][F]: strength@t changes its value from greater than 0 to 0 in $t \in [T_{start}, T_{end})$:
        issue Delete( F, UR, t)
}
```

REFERENCES

[1] Van Hook, D.J. and J.O. Calvin, *Data Distribution Management in RTI 1.3*, in *Proceedings of the Spring Simulation Interoperability Workshop*. 1998: Orlando, FL. p. paper 98S-SIW-206.

[2] A. Boukerche and A. J. Roy, "Dynamic Grid-Based Multicast Group Assignment in Data Distribution Management", Distributed Simulations and Real-Time Applications Workshop, 2000, pp. 27-34.

[3] A. Boukerche and Caron Dzermajko, "Performance Comparison of Data Distributed Management in Distributed Simulation Systems", In Proceedings of the 5th IEEE Distributed Simulation and Real Time Applications, August 2001, pp.67-75.

[4] Fujimoto, R.M., *Parallel and Distributed Simulation Systems*. 2000: Wiley Interscience.

[5] Van Hook, D.J., S.J. Rak, and J.O. Calvin, "Approaches to Relevance Filtering", in Proceedings of the 11th DIS Workshop on Standards for the Interoperability of Distributed Simulations. 1994: Orlando, FL.

[6] G. Tan, Y. Zhang and R. Ayani, "A Hybrid Approach to Data Distribution Management", IEEE Distributed Simulations and Real-Time Applications, 2000, pp. 33-40.

[7] Richard M. Fujimoto, Thom McLean, Kalyan Perumalla and Ivan Tacic, "Design of High Performance RTI Software", invited paper, 4th International Workshop on Distributed Simulation and Real-Time Applications, August 2000.

[8] Katherine L. Morse and Michael Zyda, "Multicast Grouping for Data Distribution Management", SIMPRA - Journal of Simulation Practice and Theory, Elsevier, Fall 2001.

[9] Richard M. Fujimoto, Ivan Tacic, "Time Management of Unsynchronized HLA Services", Proceedings of the Fall Simulation Interoperability Workshop, Orlando, Florida, September 1999.

[10] Ivan Tacic, Richard M. Fujimoto, "Synchronized Data Distribution Management in Distributed Simulations", Proceedings of the 12th Workshop on Parallel and

Distributed Simulation (PADS '98), Banff, Alberta, Canada, May 1998, pp. 108-115.

[11] Ivan Tacic, Richard M. Fujimoto, "Synchronized Data Distribution Management in Distributed Simulations", Proceedings of the Spring Simulation Interoperability Workshop, Orlando, Florida, March 1997, pp. 303-312.

[12] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, Daniel C. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems", International Conference on Distributed Computing Systems (ICDCS '99), June 1999.

[13] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. E. Strom, D. C. Sturman, "Exploiting IP Multicast in Content-Based Publish-Subscribe Systems".

[14] Antonio Carzaniga, "Architecture for an Event Notification Service Scalable to Wide-area Networks", Ph.D. Thesis, Politecnico di Milano, December 1998.

[15] B. Krishnamurthy, D. Rosenblum, "Yeast: A general purpose event-action system", IEEE Transactions on Software Engineering, 21(10), October 1995.

[16] Bill Segall, David Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching", Proceedings of AUUG97, Brisbane, Australia, September 1997.

[17] J. Touch, A. S. Hughes, "The LSAM Proxy Cache – a Multicast Distributed Virtual Cache", Computer Networks and ISDN Systems, 30(22-23), November 1998.

[18] L. Zhang, S. Floyd, V. Jacobson, "Adaptive Web Caching", Proceedings of the 2nd NLANR Web Cache Workshop, Boulder, Colorado, 1997.

[19] L. Zou, M. H. Ammar, C. Diot, "An evaluation of grouping techniques for state dissemination in networked multi-user games".

[20] E. Lety, T. Turletti, F. Bacccelli, "Cell-based Multicast Grouping in Large-Scale Virtual Environments".

[21] C. Kanarick, "A Technical Overview and History of the SIMNET Project," in *Advances in Parallel and Distributed Simulation*, vol. 23: Society for Computer Simulation, 1991, pp. 104-111.

[22] M. Macrdonia, M. Zyda, D. Pratt, and P. Brutzman, “Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments,” in *1995 IEEE Virtual Reality Annual Symposium*, 1995, pp. 11-15.

[23] K. Morse, “Interest Management in Large Scale Distributed Simulations,” University of California, Irvine Technical Report TR 96-27, 1996.

[24] E. T. Powell, L. Mellon, J. F. Watson, and G. H. Tarbox, “Joint Precision Strike Demonstration (JPSD) Simulation Architecture,” in *14th Workshop on Standards for the Interoperability of Distributed Simulations*. Orlando, Florida, 1996, pp. 807-810.

[25] K. L. Russo, L. C. Shuette, J. E. Smith, and M. E. McGuire, “Effectiveness of Various New Bandwidth Reduction Techniques in ModSAF,” in *Proceedings of the 13th Workshop on Standards for the Interoperability of Distributed Simulations*, 1995, pp. 587-591.

[26] T. W. Mastaglio and R. Callahan, “A Large-Scale Complex Environment for Team Training,” *IEEE Computer*, vol. 28, pp. 49-56, 1995.

[27] J. S. Steinman and F. Wieland, “Parallel Proximity Detection and the Distribution List Algorithm,” in *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, Scottland, 1994, pp. 3-11.

[28] D. J. Van Hook, J. O. Calvin, M. K. Newton, and D. A. Fusco, “An Approach to DIS Scalability,” in *Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulations*, 1994, pp. 347-356.

[29] T. D. Blanchard and T. Lake, “A Lightweight RTI Prototype with Optimistic Publication”, in *Spring Simulation Interoperability Workshop*, Orlando, Florida, 1997, pp. 551-560.

[30] Defense Modeling and Simulation Office, “High Level Architecture Interface Specification Version 1.3”, Washington D.C., 1998.

[31] R. M. Fujimoto, “The Virtual Time Machine,” in *International Symposium on Parallel Algorithms and Architectures*, 1989, pp. 199-208.

[32] Fujimoto, R.M. and P. Hoare, “HLA RTI Performance in High Speed LAN Environments”, in *Proceedings of the Fall Simulation Interoperability Workshop*, Orlando, FL., 1998.

[33] Hoare, P., G. Magee, and I. Moody, “The Development of a Prototype HLA Runtime Infrastructure (RTI-Lite) Using CORBA”, in *Proceedings of the 1997 Summer Computer Simulation Conference*, 1997. p. 573-578.

[34] Boden, N., et al., “Myrinet: A Gigabit Per Second Local Area Network”, IEEE Micro, 1995, pp. 29-36.

[35] Pakin, S., et al., “Fast Message (FM) 2.0 Users Documentation”, Department of Computer Science, University of Illinois: Urbana, IL., 1997.

[36] Ferenci, S.L., K.S. Perumalla, and R.M. Fujimoto, “An Approach for Federating Parallel Simulators”, in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, IEEE Computer Society, 2000, pp. 63-70.

[37] Jefferson, D. R., “Virtual Time”, *ACM Transactions on Programming Languages and Systems*, 7(3), pp. 404-425.

[38] Chandy, K. M., and J Misra, “Distributed Simulation: A Case Study in Desing and Verification of Distributed Programs”, *IEEE Transactions on Software Engineering*, 5(5), pp. 440-452.

[39] Fujimoto, R. M., D. M. Nicol, “State of the Art in Parallel Simulation”, in *Proceedings of the Winter Simulation Conference*, 1992, pp. 122-127.

[40] Lubachevsky, B. D., “Efficient distributed Event-driven Simulations of Multiple-loop Networks”, *Communications of the ACM*, 32(1), pp. 111-123.

[41] George Riley, Mostafa Ammar, Richard Fujimoto, Alfred Park, Kalyan Perumalla, and Donghua Xu, “A Federated Approach to Distributed Network Simulation”, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Vol. 14(2), April 2004.

[42] George F. Riley, “The Georgia Tech Network Simulator”, *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, August 25-27, 2003, Karlsruhe, Germany.

[43] Richard Fujimoto, et al, “F*DK User Guide”*, College of Computing, Georgia Institute of Technology, February 2001.

[44] O. Hagsand, *“Interactive MultiUser VEs in the DIVE System”*, IEEE Multimedia Magazine, Vol 3, Number 1, 1996.

[45] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe, *"Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications"*, 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1999.

[46] D. J. Van Hook, S. J. Rak, and J. O. Calvin, *"Approaches to RTI implementation of HLA data distribution management services"*, Proceedings of the Spring Simulation Interoperability Workshop, Orlando, Florida, 1996.

[47] Ayani, *"A parallel simulation scheme based on the distance between objects"*, Proceedings of the SCS Multiconference on Distributed Simulation, 21(2):113-118, March 1989.

[48] David M. Nicol, *"The Cost of Conservative Synchronization in Parallel Discrete Event Simulations"*, J. ACM 40(2): 304-333, 1993.

[49] A. Gafni, *"Rollback mechanisms for optimistic distributed simulation"*, Proceedings of the SCS Multiconference on Distributed Simulation, pages 61–67, 1988.

[50] Darrin West, *"Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation"*, Master Thesis, University of Calgary, 1988.

[51] M. Sokol, D. P. Briscoe, and A. P. Wieland, *"MTW: a strategy of scheduling discrete simulation events for concurrent execution"*, Proceedings of the SCS Multiconference on Distributed Simulation, 19, 3, pages 34-42, 1988.

[52] F. Reynolds, *"A spectrum of options for parallel simulation"*, In Proceedings 1988 Winter Simulation Conference, pages 325–332, 1989.