Usability Enhancements for the DVERT Simulator Architecture

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

____

James Montgomery

Date: January 12, 2011

Approved:

____

Professor George Heineman, Major Advisor

____

Professor Hugh Lauer, Co-Advisor

____

Professor Edward A. Clancy, Co-Advisor

This work was sponsored by the Department of the Air Force under contract number FA8721-05-C-0002. Opinions, interpretations, recommendations and conclusions are those of the authors and are not necessarily endorsed by the United States Government.

Abstract

This paper describes the continuation of an earlier initiative to develop a distributed, real-time radar simulator at MIT Lincoln Laboratory. The project consisted of updating the existing simulator code developed during the previous project to eliminate race conditions and to allow it to function in a distributed setting. In addition to identifying and eliminating race conditions, the middleware layer of the simulator was also reconfigured and tested in order to enable distributed communication between multiple machines. The current project required a substantial amount of debugging and testing in order to identify the issues present in the existing simulator and create solutions to address these problems.

Executive Summary

This project describes the continued development of a distributed, real-time radar simulator created for MIT Lincoln Laboratory. The current work on the simulator focused largely on synchronizing the simulation engine, which is responsible for delivering events and message passing between processes. The simulator runs on a platform that contains multiple processes, each of which contain multiple threads. All of these separate processes and threads must be coordinated in order to ensure that the simulator runs deterministically and produces correct output.

The first major component of this project was to research, plan, and implement a capable simulation timing system to enforce order between scheduled events. This simulation timing system was used as a means to schedule events across multiple processes without worrying that one process of the simulator would get ahead of another and execute events out of order. Special timing values called "look-aheads" were sent with each event, indicating the minimum time that another event of that type would be expected. This allowed different processes of the simulator to stay coordinated and know when different event types would be arriving from external simulator processes.

In the second portion of the project, the critical sections of the simulator execution cycle were secured and the simulation engine was made thread-safe. The previous execution cycle of the simulator engine was subject to race conditions due to the fact that events could be inserted into the main event queue at any time when the simulator was running. The simulation engine was hence changed so that incoming events were pipelined into a queue, which was emptied at the beginning of every execution cycle. The queue was specifically designed to be thread-safe, allowing multiple event listener threads to safely add events to a single container.

A large portion of the work required of this project involved debugging the simulator output and determining the cause of event disorder and race conditions. Several test scenarios were used in order to verify the stability of the simulator once all revisions had been made. Upon completion, the work was presented to the laboratory formally and a final version of the simulator code was synchronized to the laboratory repository. The final revision allows for exceptionally stable simulation across multiple processes on a single machine. The work done in this project greatly increases the stability of the simulator in development, as well as positioning it to be used for truly distributed computation in future developments.

Acknowledgements

The work on this project was facilitated by the help of two main advisors, Professors Hugh Lauer and Edward Clancy, who oversaw the work on this project. Unlike the previous work performed on the simulator, the developments performed in this project were performed outside of Lincoln Laboratory with a single member of the previous project team while other academic classes were being taken. This slowed progress significantly, but flexibility from and support from both the advisors and the Laboratory ensured that progress was made in advancing the simulator forward.

Regular visits to the laboratory were made in order to perform testing and report back to the Lincoln Laboratory staff. They were particularly helpful in providing support for configuring the Rosa Thin Communications Layer (RTCL), which was one of the central pillars allowing the simulator to perform in a distributed setting. Above all, their flexibility with the project was most appreciated considering the high demands of other academic classes that were taken during the project duration. A special thanks goes out to Seth Hunter and the rest of the Lincoln Laboratory staff.

Finally, the continued work on this project would not be possible if not for the efforts of the original project team, including partners Matthew Lyon and Lucas Scotta. The amount of work put into the original endeavor is what led to the making of an exceptional piece of software that the Laboratory will continue to develop into the future.

Contents

Abstract 2
Executive Summary 3
Acknowledgements 5
Table of Figures 8
1. Introduction 9
1.1 Original Simulator Project Work 9
1.2 Project Description 10
1.2.1 Simulator Timing 10
1.2.2 Securing Critical Sections 10
1.3 Importance of the Project 11
2 Background 11
2.1 Distributed Computing 11
2.1.1 Benefits of Distributed Computation 11
2.1.2 Issues with Distributed Processing 12
2.1.3 Synchronization of a distributed system 13
2.2 ROSA Thin Communications Layer 14
2.3 Previous DVERT Simulator Implementation 15
2.3.1 Simulator Architecture 16
2.3.2 Distributed Outline 17
3 Methodology 18
3.1 Resources and Tools 18
3.2 Project Outline 19
3.3 Metrics for Success 20
4 Debugging and Implementation 21
4.1 Identifying Existing Problems 21
4.2 Simulation Timing 22
4.2.1 Adding Look-Aheads and External Timers 23
4.2.2 Multiple environment simulators 25
4.2.3 Enforcing Event Causality 26
4.3 Securing Critical Sections 27

4.3.1 Initial Execution Order 28
4.3.2 Revised Execution Order 29
5 Results and Analysis 30
5.1 Basic Testing Scenarios 30
5.2 Advanced Testing Scenarios 31
6 Conclusion 32
6.1 Summary 32
6.2 Outstanding Issues 32
6.3 Future Work 32
6.3.1 Middleware Configuration 33
6.3.2 Enhanced Models 33
6.4 Concluding Thoughts 33
References 35

Table of Figures

Figure 1 – Amdahl's Law 12
Figure 2 - The flow of data through the DVERT simulator. Each block represents a separate process. 16
Figure 3 - Message passing between models through the use of events... 17
Figure 4 - The distributed outline for the DVERT simulator 18
Figure 5 - A table of events and actions for a basic simulator timing system 23
Figure 6 - A table of events and actions for the updated simulator timing system 25
Figure 7 - The problem with assuming events arrive in order. 26
Figure 8 - Multiple queues should be used to store the event times for received events......................... 27
Figure 9 - The original DVERT execution order 28
Figure 10 - The revised execution order, which secures all previous critical sections. 29
Figure 11 - The table of initially tested simulator configurations.. 30
Figure 12 - Advanced tests used to test the simulator output 31

1. Introduction

Radar simulation is a common practice used by MIT Lincoln Laboratory to test and debug radar processing software. This technique uses software to simulate the radar's physical hardware and the surrounding environment, producing realistic returns which are sent to the radar processing software. Group 33 is currently in the process of phasing out a legacy radar simulator and replacing it with a new distributed alternative, designed to be scalable to simulate a large number of radar targets. However, some work remains before the new simulator, named DVERT[1], can be used practically by the group.

1.1 Original Simulator Project Work

At the end of the last development cycle for the new simulator, the program was capable of radar simulation distributed across multiple processes, with one process being used to simulate the radar hardware and multiple processes used to simulate the surrounding targets and environment [Montgomery, Lyon, Scotta, 2010]. The returns produced by the simulator were accurate when compared with the original mathematical equation which they were modeled upon. Multiple targets could also be accurately simulated within multiple environment simulator processes, though the simulator was subject to frequent crashes and very limited stability.

Due to the time constraints of the previous development cycle, there were some outstanding issues with the new simulator implementation that were unable to be addressed. The first of these is a synchronization issue within the simulator where messages are occasionally passed out of order, creating a race condition. This is the main deterrent to allowing the simulator to run in real time, which is required for the simulator to interface with the real-time radar processing software at the Laboratory.

[1] DVERT is an acronym for "Distributed Virtual Environment for Radar Testing"

Additionally, the simulator middleware needs to be configured in order to execute over the network in a true distributed fashion. In order to acquire parallel performance gains within the simulator, the middleware layer must be configured and tested over multiple machines in a distributed setting.

1.2 Project Description

The main goal of this project is to bring the DVERT simulator implementation into a usable state for Lincoln Laboratory. The two main objectives detailed in this project are displayed below:

1.2.1 Simulator Timing

The existing processes used within the simulator needed to be coordinated in order to eliminate race conditions between processes. This involved debugging the timing system within the simulator implementation to ensure that messages and events are being received in the correct order. Some of the timing system for the simulator was implemented towards the end of the original simulator project, so the current project involved testing existing components as well as coming up with new solutions in areas which had not yet been fully implemented.

1.2.2 Securing Critical Sections

Even within a single process there are multiple threads and critical regions that must be managed in order to produce deterministic output. Simulator stability should be ensured by making sure that only one thread can enter these regions at a time. In addition, thread-safe containers should be used to guarantee deterministic output when interacting with data objects.

1.3 Importance of the Project

Group 33 (Ranges and Test Beds) of the Laboratory has expressed great interest in using DVERT within its everyday operations for simulation. The group performs frequent radar simulations, especially before fielding new hardware and performing physical flight tests. A distributed simulator would give the Laboratory the ability to perform higher fidelity simulations involving larger numbers of targets that require great amounts of computation. Ideally, a distributed simulator would allow the Laboratory to increase the processing ability of the simulator simply by adding additional machines to the distributed network. This project recognizes the utility of such a simulator and strives to bring the DVERT implementation to a usable state where it can be passed off to the Laboratory.

2 Background

This section outlines the domain-specific knowledge required to make progress on this project extension. Considering the work is largely in the general realm of simulation, no radar-specific background was necessary. Background regarding the radar specific components of the simulator can be found within the previous project report [Montgomery, Lyon, Scotta, 2010].

2.1 Distributed Computing

One of the primary goals of the DVERT simulator architecture is to be able to perform simulation using "distributed" computation. This means that the processing for a single simulation will be divided and performed by multiple machines; the workload is divided in parts in order to leverage the additional processing power of using multiple machines to accomplish one larger task.

2.1.1 Benefits of Distributed Computation

The benefits of distributed computing are explained succinctly by Amdahl's Law [Amdahl, 1967], a mathematical equation displaying the amount of speedup that can be obtained in a computer program where tasks can be performed at the same time, or "in parallel." The relationship derived by Amdahl is shown in Figure 1:

$$Speedup = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}$$

Figure 1 – Amdahl's Law

This equation describes a system with *N* processors, which spends *s* time on serial tasks and *p* time on parallel tasks. "Serial" tasks must be performed in order, and cannot be divided and processed simultaneously. As the number of processors included in the system increases, the term $\frac{p}{N}$ approaches zero and the sole factor that limits the speedup of the system is the amount of time spent on serial tasks. Hence by leveraging multiple processors by using multiple machines, we can significantly decrease the amount of time it takes to perform computation—to the extent that the computation can be divided up into many parallel tasks.

2.1.2 Issues with Distributed Processing

Despite the mentioned benefits of distributed computing, it can also lead to some issues that adversely affect the outcome of the overall processing being performed. One of these issues is the creation of "race conditions," or situations where the outcome of processing a series of tasks is uncertain since the tasks may be performed out of order when executed in parallel. To give an example of this, consider the scenario below:

Variable A initialized to 2
Process 1 will set A = A+5
*Process 2 will set A = A*5*

In this scenario we have a variable in shared memory named A, which is initialized to 2. Two processes act on A in parallel, performing different operations. No particular order is imposed on the way these processes execute, so either process 1 or process 2 may perform its operation first. This leads to several possible outcomes depending on the order in which events occur:

Scenario 1
Original value of variable A = 2
Process 1 will set A = 2+5= 7
*Process 2 will set A = 7*5 = 35*
Final value of A = 35

Alternatively, a second situation is also possible:

Scenario 2
Original value of variable A = 2
*Process 2 will set A = 2*5= 10*
Process 1 will set A = 10+5 = 15
Final value of A = 15

This ambiguity in the processing sequence causes serious issues when attempting to reliably perform computation on a distributed platform. Most applications rely on the ability to reliably perform computations in a specific order. A distributed application should strive to compute as many calculations in parallel as possible, while still enforcing the order necessary within the overall program.

2.1.3 Synchronization of a distributed system

In order to enforce order during distributed computation, "synchronization" should be performed at necessary stages of computation. During synchronization, the results of tasks executed in parallel are organized or combined to produce a consistent, correct result before a serial task must be performed. For example, in a situation where 50 parallel calculations can be performed and combined before a particular serial calculation, the speed of all calculations can be optimized as follows:

50 separate threads simultaneously execute the 50 parallel tasks
The main process waits for all threads to finish calculating
The main process combines all results
The main process performs the serial calculation

This type of synchronization is referred to as “barrier synchronization,” since the program effectively creates a barrier where computation cannot proceed to the serial task until all the parallel tasks are finished executing. In this way, the overall order of the program is enforced since the parallel tasks (which can be performed in any order) are all computed and organized before combining their result with the next serial computation. The speedup gained from using multiple processors can still be leveraged despite the fact that specific tasks in the program must be performed in a particular order.

2.2 ROSA Thin Communications Layer

The ROSA Thin Communications Layer is proprietary software developed by MIT Lincoln Laboratory. Abbreviated as RTCL, the software acts as an abstract communications wrapper used for communicating between different processes. Effectively, the program creates a standard for communicating between different processes that is agnostic towards the actual transport mechanism used to move the data from one process to another.

RTCL supports several transport layers for inter-process communication, such as local communication over shared memory and communication over the network using a commercial network transport called RTI-DDS. This support allows users to create programs that can execute between processes either locally or over a network simply by changing a few lines in an RTCL configuration file. This has great appeal in developing distributed systems which may need to support an arbitrary number of machines networked together while alternatively allowing testing to be performed on a single machine through shared memory.

RTCL is configured using an XML file that is read in before the start of communication. Editing this file allows a developer to change the transport layer used, the number of clients allowed, as well as factors such as the IP range being serviced when performing over the network. RTCL was used as the middleware layer in the simulator due to its applicability to distributed computing and its ready availability from the Laboratory.

2.3 Previous DVERT Simulator Implementation

The DVERT simulator performs radar simulation based on a model used in an existing "legacy" simulator created at the laboratory. In the DVERT system, simulation begins with the radar control program, which configures the simulated radar hardware by sending it a brief message referred to as a "UCM" (Universal Control message). From here, the simulated hardware sends simulated pulses of radar energy to simulated targets in a separate environment simulator. These pulses produce energy "returns," which represent energy that bounces off the target and is detected by the radar receiver. These returns are sent back to the simulated radar hardware and then to the radar control program for display and further processing. A diagram explaining the flow of information in the DVERT simulator is displayed below:

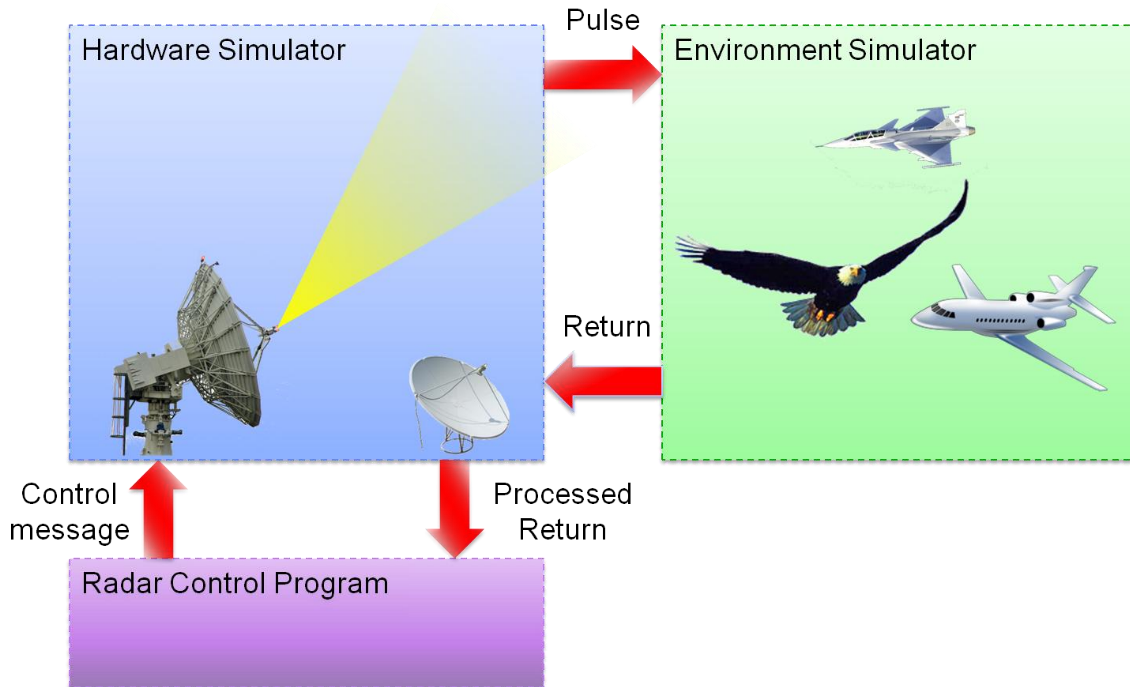

Figure 2 - The flow of data through the DVERT simulator. Each block represents a separate process [Montgomery, Lyon, Scotta, 2010].

2.3.1 Simulator Architecture

The previous DVERT simulator was based on a simulator architecture referred to as OASIS (Open Architecture Simulation Interface Specification) developed at MIT Lincoln Laboratory [MIT Lincoln Laboratory, 2009]. This architecture allows all of the domain specific information regarding radar to be contained in a set of classes called "models," which are totally agnostic towards the simulation "engine." While the models perform all of the domain specific radar calculations, the simulation engine organizes a queue of messages called "events" that pass data back and forth between different models and other simulation engines.

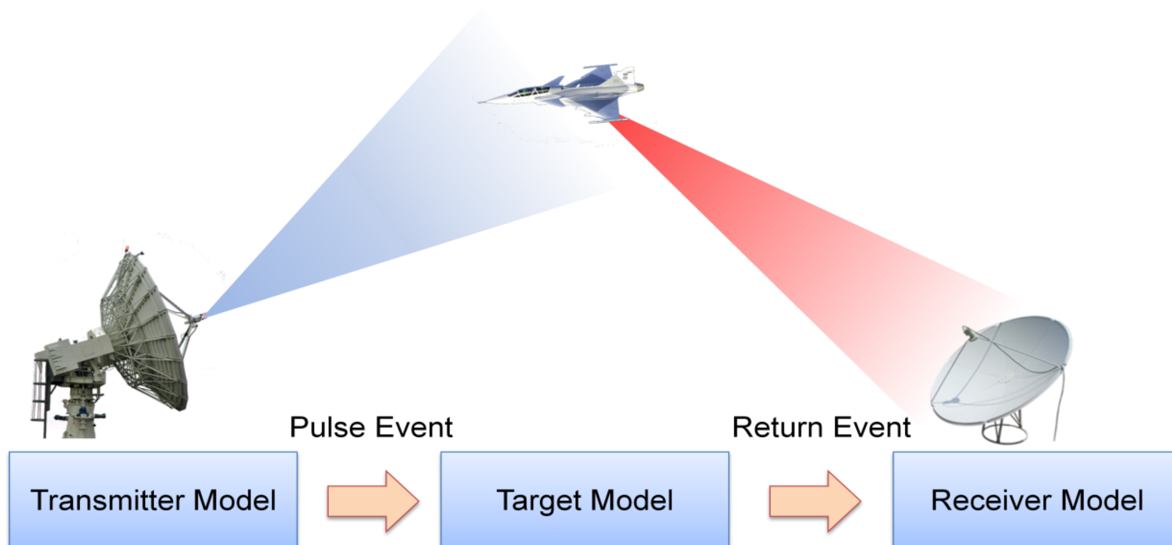

Figure 3 – DVERT simulator message passing between models through the use of events [Montgomery, Lyon, Scotta, 2010].

This separation of domain knowledge and general simulation implies that the DVERT simulation could be paired with a different set of models and simulate an entirely different domain.

2.3.2 Distributed Outline

The DVERT distributed outline is designed to parallelize the large amount of calculation that must be performed in computing the returns of radar pulses as they hit targets in the environment. These returns must be calculated for each individual target and for each separate radar pulse which emanates through the environment. Considering there may be as many 2000 pulses per second and an arbitrary number of targets, a significant amount of computation must be performed in order to calculate the final energy return which is sent back to the receiver.

Fortunately, the order in which these returns must be calculated is arbitrary. That means that all of these returns may be calculated in parallel, then added up to produce the final energy return that is sent to the radar receiver. Considering the parallel speedup that was discussed in Section 2.1.1, we can note the tremendous benefit that we can leverage by performing parallel processing in order to calculate these returns. An outline of the DVERT distributed architecture is displayed in Figure 4:

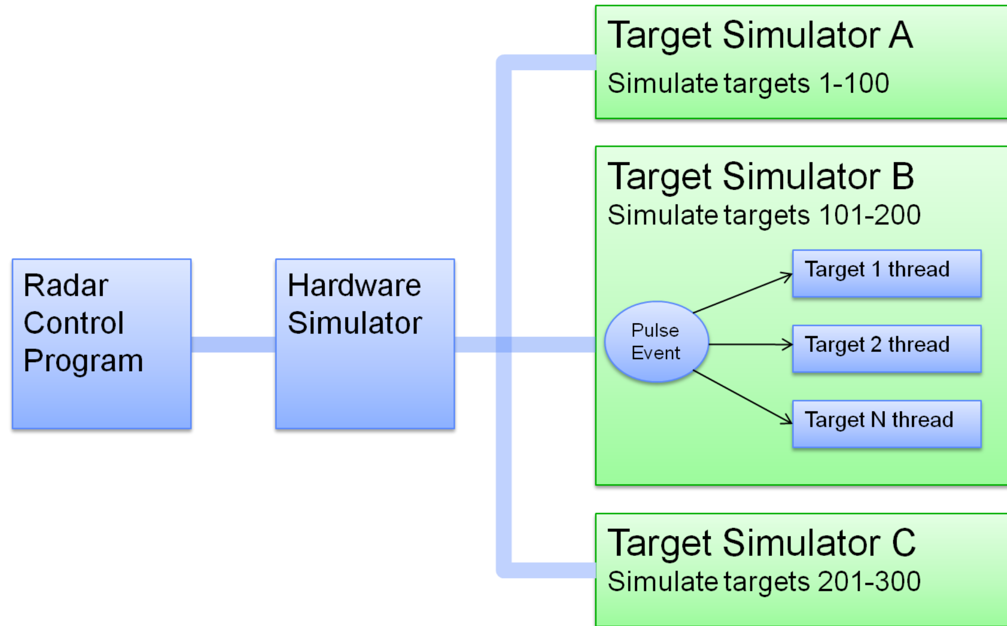

Figure 4 - The distributed outline for the DVERT simulator [Montgomery, Lyon, Scotta, 2010]

The simulator uses a single process to represent the hardware present in a radar system, while utilizing multiple processes with multiple threads in order to represent the targets within the radar's environment. The targets are distributed among the separate Target simulator processes. Pulses are sent from the radar hardware to each of the separate target simulator processes. These pulses trigger return calculations to be performed by all targets within each target simulator.

Once all of these returns are calculated, they are then summed and returned as a single set of values to the hardware simulator.

3 Methodology

3.1 Resources and Tools

There are multiple resources and tools that were necessary in order to complete this project. The foremost of these resources were the experts at Lincoln Laboratory within Group 33 who have

information regarding the legacy radar control program used by the Laboratory. They also had information for configuring the middleware layer of the simulator that will expedite getting the simulator set up over the network once it is stable. Due to a lack of transportation, visits to the Laboratory were infrequent throughout this project. The project schedule was designed around this limitation, with visits being consolidated as much as possible and the majority of the work and testing being performed off of the Lincoln Laboratory campus.

The software tools required for the project included the Eclipse IDE [The Eclipse Foundation, 2010], the CxxTest Unit Testing Suite [Fitch, 2009], as well as the Subversion version control system [Apache Software Foundation, 2010]. There were also several software components specific to the Laboratory, including the radar control software (referred to as ROSA II[2]), as well as the abstract middleware wrapper used in the simulator (referred to as RTCL[3]).

3.2 Project Outline

In order to accomplish the objectives stated in this project, a plan was outlined detailing the course of the project over its seven-week timespan. Additionally, the necessary resources and tools required for the project were determined. Finally, the metrics for determining the success of the project were decided upon.

The outline for each week of the project is displayed below:

- **Week 1 –** Start debugging timing problems with the latest simulator implementation. This can be performed and tested without needing to physically go to Lincoln Laboratory. Identify race conditions and issues with the current timing system.

[2] ROSA is an acronym for "Radar Open Systems Architecture"
[3] RTCL is an acronym for "ROSA Thin Communications Layer"

- **Week 2-** Begin implementing fixes to the simulator timing system. Plan and proof a timing system that can work across multiple processes.
- **Week 3 –** Finish implementing the timing system. Perform basic acceptance tests to verify proper performance.
- **Week 4 –** Begin securing internal critical sections. Create appropriate thread-safe containers. Organize program execution order.
- **Week 5-** Finish securing critical sections for use with multiple threads. Complete reorganization of program execution order.
- **Week 6 –** Perform testing of the simulator under different scenarios. Verify that satisfactory results are achieved, events are in order, and no returns are dropped.
- **Week 7-** Finalize a release version of the simulator code for presentation to the Laboratory. Organize documentation of changes that have been made.

3.3 Metrics for Success

In order to determine the success of this project, clearly defined metrics were decided for each of the primary objectives. These metrics are listed below:

- **Perform successful simulations using five different scenarios** – In order to test that the timing issues with the simulator have been resolved, the simulator will be configured with five varying scenarios to be tested against. These scenarios will be run for at least 10 minutes in total, for at least three trials. This metric will be met if the returns produced by the simulator are correct during the timeframe of each trial. The output of the simulator will be piped into a file so it can be checked for errors after the simulation is completed.
- **Perform a multi-process simulation for more than 24 hours** – In order to verify that all race conditions and critical sections have been managed, it is necessary to test running the simulator

for an extended amount of time. If a multi-process simulation can be performed for a significant length of time, it provides evidence that issues such as race conditions that would lead to indeterminate behavior and crashes are not present. It also reinforces confidence in the stability of the simulator when using multiple target simulators.

4 Debugging and Implementation

The breadth of work required of this project was in performing the massive amount of debugging and testing required to characterize the problems occurring with the existing simulator implementation and properly address them. The various problems that were solved throughout debugging and testing are detailed in the following sections.

4.1 Identifying Existing Problems

One of the most prominent problems with dealing with the existing simulator code was the evidence of race conditions from the simulator output. Particularly, returns being sent back to the hardware simulator were appearing with empty values when these returns should have contained the energy returns from the multiple targets in the environment. After testing the models and ensuring that all the radar-specific calculations were being performed correctly, race conditions between the different simulator processes were determined to be the cause of the problem.

The first step in solving the problems with the simulator output was characterizing the symptoms of the race condition. Viewing the output of the simulator after a large number of tests showed that the individual returns of each target were being calculated and added correctly, but the aggregate of these returns referred to as a “receive window” was occasionally coming back to the hardware simulator with

blank values. This implies that the returns being calculated by the target simulator were not successfully being added into the receive window and being sent back to the hardware simulator; a key timing constraint within the simulator was not being met. The time occurrence of these blank returns was random and could not be reproduced consistently through multiple trials of running the simulator with the same configuration. Debugging further, it was found that some returns were being dropped; receive windows were being sent back to the hardware simulator before the actual returns from targets were being added to the window. Resolving this race condition between the calculation of the returns and when receive windows were returned was the first step to allowing the simulator to execute in a deterministic manner.

4.2 Simulation Timing

A timing system was implemented in this project in order to prevent this race condition from occurring, as well as to reinforce the overall order of the simulator. Partially implemented in the existing simulator code, it was expanded and made complete through further testing and debugging done in this project. In this system, every event is given a “Simtime,” or a simulation time in which it is scheduled to execute. In this way, every event is effectively assigned an order that it will execute in; when all the events for one simulation time are complete, the simulator will then advance the clock and process events for the next moment in simulation time.

The table in Figure 5 illustrates this simple event system. Starting at Simtime 0, it advances the time to the soonest event. It then processes all the events for that time, and advances the simulation time to that of the next lowest event. This continues until there are no remaining events, upon which the simulation time is held constant:

Event and Time	Next Simulator Action
No events until Simtime 2	Advance Simtime to time 2
Pulse Event A at Simtime 2	Process Pulse Event A
Pulse Event B at Simtime 3	Advance Simtime to 3. Process Pulse Event B.
Return Event A at Simtime 3	Process Return Event A
Return Event B at Simtime 4	Advance Simtime to 4. Process Return Event B and hold Simtime at time 4

Figure 5 - A table of events and actions for a basic simulator timing system

While this simulation timing system works well when all events are known, it fails in a system where there are external events coming from external processes. Considering that the simulator is composed of multiple processes, keeping everything synchronized requires additional restraints on the timing system.

4.2.1 Adding Look-Aheads and External Timers

In the previous section an outline for a basic simulator timing system was described that enforced order among *local* events being processed by the simulator. However, issues with this system arise when *external* events arrive from another simulator that must be processed. It is impossible to determine whether or not it is safe to advance the *local* simulation time of one process without knowing when events will be arriving from external processes. In order to overcome this problem, a system of external timers and event “look-aheads” was implemented [Fujimoto, 2000].

External timers are used to store the time at which events from external processes are scheduled. For example, the target simulator may have an external timer for pulse events coming from the hardware simulator in order to determine when it is expecting a pulse event. This effectively prevents receive windows from leaving the target simulator with empty values, because it forces the pulse events to be processed first and immediately causes scheduling of return events which are added to the receive window. By having a process keep external timers for each other external process and the event types associated with them, it is possible to enforce order across a large breadth of processes that would otherwise not be possible. This still leaves one lingering problem, however; how will these timers be set? How can a process know when it should and should not be expecting an external event?

This problem can be compactly solved through the use of "look-ahead" values for each event that is sent. A look-ahead is a unit of time that specifies the minimum amount of time in which the process that sent an event will send another event of that type. For example, if the hardware simulator sends a pulse event to the target simulator at time 2 with a look-ahead of 3, it guarantees that it will not send another pulse event until at least time 5. Now, the target simulator can safely advance to time 5 by knowing that it will not receive any external events during this period.

Look-aheads and external timers must be used for each unique combination of external process and event-type in order to enforce order. In the event that multiple event-types are received from external processes, the current process can only advance Simtime as far as the lowest scheduled event+look-ahead value. An example of the updated timing system using external timers and look-aheads is displayed in figure 6:

Hardware Simulator	Target Simulator External Timer	Next Target Simulator Action
Pulse Event A sent at Simtime 0, look-ahead 1	External timer for pulse events: 0	Process pulse event A. Set external timer for pulse events to 1
Pulse event B sent at Simtime 1, look-ahead 2	External timer for pulse events:1	Process pulse event B. Set external timer for pulse events to 3
Pulse event D sent at Simtime 4, look-ahead 1	External timer for pulse events: 3	Wait at time 3. Cannot advance to time 4 since the external timer is at time 3.
Pulse event C sent at Simtime 3, look-ahead 1	External timer for pulse events: 3	Process pulse event C. Set external timer for pulse events to 4.
No events sent	External timer for pulse events: 4	Process pulse event D. Set external time for pulse events to 5.

Figure 6 - A table of events and actions for the updated simulator timing system

4.2.2 Multiple environment simulators

Using multiple target simulator processes over a distributed platform is the ultimate goal for the DVERT simulator architecture. In the same way that external timers can be used with different external event types, they can also be used to enforce order when these events originate from different sources. For example, if simulator A and simulator B both send pulse type events to simulator C, the order of the events between every simulator can be dictated by using a separate timer for each unique event-type and source pair. In this case, simulator C would have two external timers; both of them keeping track of pulse events, but distinguished by the ID of the sender.

4.2.3 Enforcing Event Causality

Even with the presence of look-aheads and external timers in each simulator process, there is still one final component required to enforce rigid event ordering between each process. This was the modification of external timers to use priority queues in which groups of external times from events could be stored. Priority queues were used as they automatically order their contents by a specified value. In this case we order the queue by simulation time, ensuring that the lowest times are readily accessible at the front of the queue.

The lowest of the event times in this queue represents the minimum external time for this event type and source combination. Using a priority queue to store event times guarantees that no events are overlooked, even if events arrive out-of-order. The table below characterizes the problem caused by assuming that events arrive in the order that they are sent:

Events Arriving	Target Sim External Timers	Target Sim next action logic
Pulse event B arrives: Time 3, look-ahead 2 **UCM event A arrives: Time 2, look-ahead 10**	Update Pulse Event timer from 0 to 5 Updated UCM event timer from 0 to 10	Lowest external timer value is now 5 (from pulse event). It is now safe to advance to time 2 and process the UCM event.
Pulse event A arrives: Time 1, look-ahead 2	No external timers are changed. 3 is already lower than 5, the value of the pulse event timer	Discard the received pulse event A. It occurs in the past. Only process pulse event B.

Figure 7 - The problem with assuming events arrive in order.

To remedy this problem, a queue can be used to store the times of each event as it arrives. After an event is processed, its time is removed from the external time queue for that specific event type/source pair. A diagram of this system is displayed below:

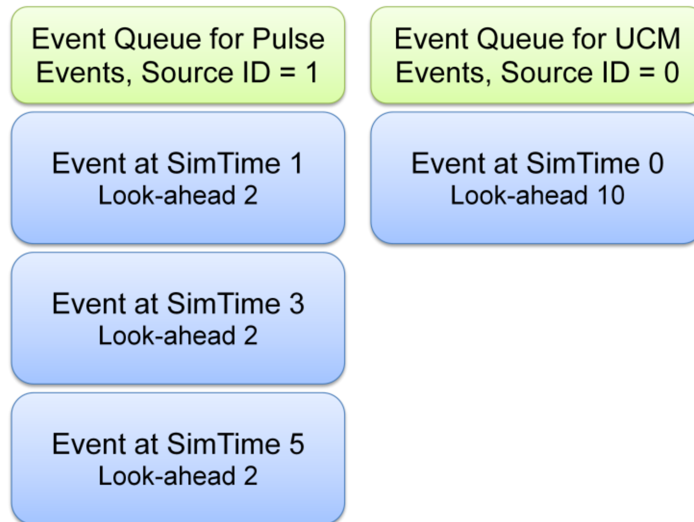

Figure 8 - Multiple queues should be used to store the event times for received events until they are processed. When an event is processed, its time is removed.

If an event arrives out of order, its timing information will simply sit in the queue until the event which precedes it is processed. In this way, events at lower times must always be processed before later events, even if the later event arrives first.

4.3 Securing Critical Sections

Even with an effective simulation timing system in place, the multi-threaded nature of the DVERT simulator requires critical sections to be secured in order to produce deterministic output. Many debugging sessions resulted in the simulator crashing or hanging up when run for extended periods of time. In order to ensure the stability of the simulator when run for longer periods of time, these critical sections needed to be identified and secured so that they could only be accessed by a single thread at one time.

4.3.1 Initial Execution Order

The standard execution order of the DVERT simulation engine before starting this project is displayed below. It consists of a main thread which executes events in the main event queue, as well as listener threads for each event type expected by the simulator:

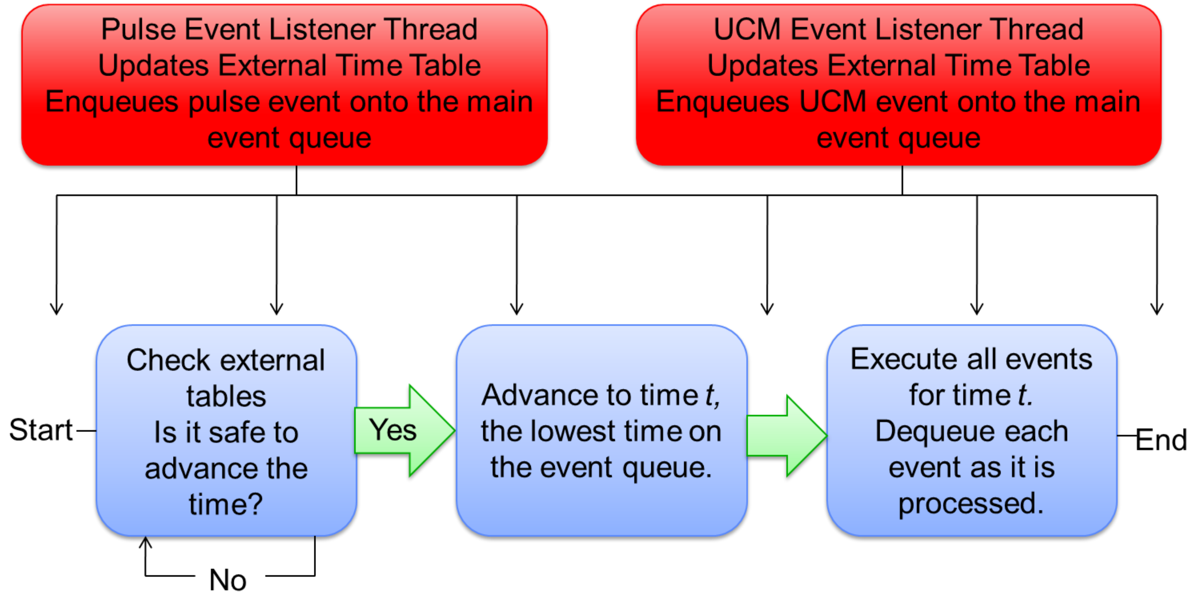

Figure 9 - The original DVERT execution order

In the main execution loop, the simulation engine performs several simple tasks. First, it checks the external time tables to see if it is safe to advance the current simulation time. If this is true, the time is advanced and all the events for the new time *t* are processed. As each event is processed, it is dequeued. This loop is repeated indefinitely.

As this loop is running, several listener threads run in the background that listen for incoming events of each type. As they are received, these threads directly update the external time tables as well as enqueue the received event into the main event queue. The pointed arrows coming from these threads across the breadth of the simulation diagram imply that these actions could occur at any time during the main processing loop.

The fact that multiple threads could be simultaneously accessing the external time tables as well as the main event queue presents a large problem. Properly securing these sections and organizing the execution loop is central to avoiding race conditions and ensuring reproducible simulator output.

4.3.2 Revised Execution Order

In response to the race conditions present in the previous execution order, a new execution order was created. This order is summarized by Figure 10:

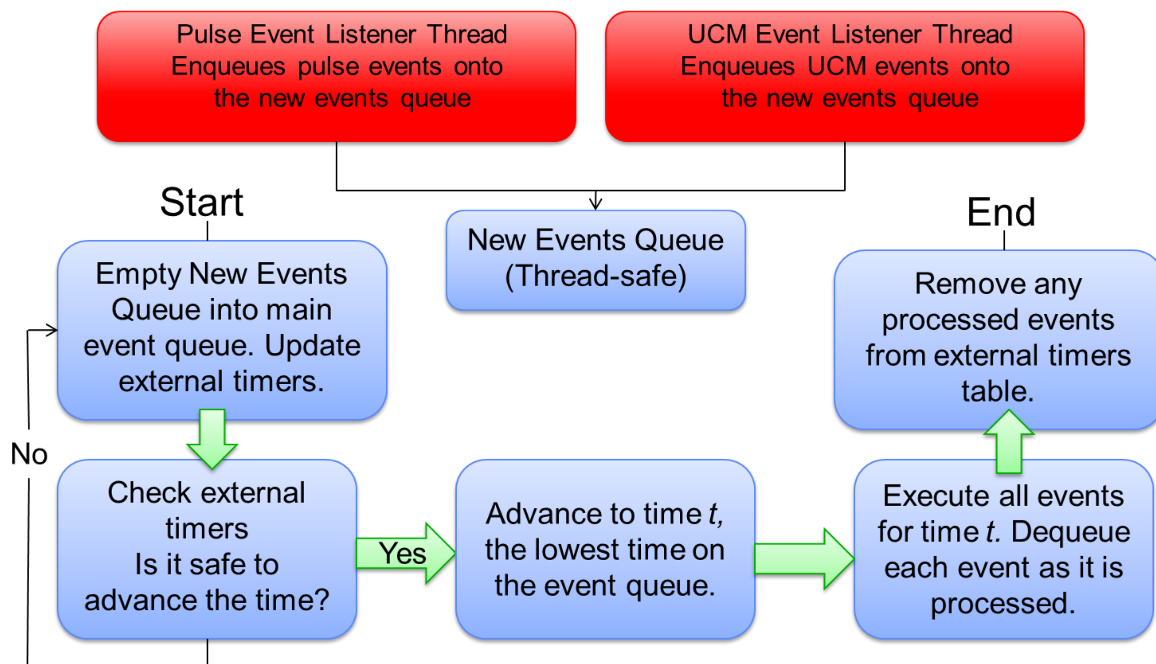

Figure 10 - The revised execution order, which secures all previous critical sections.

In this system, all event listener threads empty the events they receive into a separate, thread-safe queue called the “New Events Queue.” The Boost Thread library was used to create a suitable queue that could be locked and only used by a single thread at a time. At the start of the main execution loop, the New Events Queue is emptied into the main event queue in the engine. This creates an organized and secure point at which new events can enter the simulation engine.

From here, the simulator executes similarly to the previous model. The external timers are checked to ensure that it is safe to advance the time to the next event in the main event queue. If so, the time is advanced and all the events for this time are executed. The events are popped and their corresponding entries in the external time table are removed in the situation that the event processed was an external event. This loop repeats constantly while the simulator is running.

5 Results and Analysis

The results of this project correspond to the original metrics of success defined for this project. The first metric challenged the simulation engine to perform correctly under five different scenarios when executed for durations of at least 15 minutes. In all tested scenarios, one hardware simulator was used (as recommended within the DVERT specification), with the potential for multiple targets and target simulators. Three trials were performed per scenario, with a script checking output files for proper event order and evidence that no returns were dropped.

5.1 Basic Testing Scenarios

Five basic scenarios were defined in order to meet the standards of the first metric for success. A table characterizing these initial tested scenarios is shown in Figure 11:

Number of Target Simulators	Number of targets per target simulator	Nature of targets	Run-time per trial
1	2	1 stationary, 1 moving	30 minutes
1	100	50 stationary, 50 moving	30 minutes
2	50	25 stationary, 25 moving (per simulator)	30 minutes
2	100	50 stationary, 50 moving	30 minutes
3	100	50 stationary, 50 moving (per simulator)	30 minutes

Figure 11 - The table of initially tested simulator configurations

In every trial of these scenarios, a script was used to verify that no returns were dropped between the hardware and target simulators and that no events were processed out of order. Even for larger numbers of targets and target simulators, this was found to be true.

5.2 Advanced Testing Scenarios

Having satisfied the initial testing scenarios, two additional tests were performed as detailed in Figure 12:

Number of Target Simulators	Number of targets per simulator	Nature of targets	Run-time per trial
1	10000	5000 stationary, 5000 moving	30 minutes
3	100	50 stationary, 50 moving	24 hours

Figure 12 - Advanced tests used to test the simulator output

In the first test, a massive number of targets were simulated within one target simulator. No returns were dropped and the event order was found to be correct, but one issue was detected. A large amount of energy was returned from the targets due to their significant numbers, sometimes overflowing the "Short" integer value used within the receive-window gates in the models portion of the simulator. This can be fixed by using larger data types for each gate (int, or long int), or by scaling down returns by a constant factor as they are received (referred to as "manual gain control").

In the second test, multiple target simulators were used to perform a simple simulation for a lengthy period of time. Three target simulators were used to simulate 100 targets each for an entire day. This test was specifically designed to test for race conditions and obscure crashes that may go undetected in shorter tests. Due to the length of this test, it was only performed for a single trial. After checking the output of this trial, it was observed that no returns were dropped and all events progressed in the correct order.

6 Conclusion

In this project, great steps were made towards increasing the stability and performance of the latest DVERT Simulator implementation. The outcome of the project as well as related future work is detailed within this section.

6.1 Summary

The DVERT distributed simulator architecture is a powerful software framework which could be of great use to MIT Lincoln Laboratory in the future. This project has corrected some core issues with the engine of the simulator that prevented its usability and stability during execution. With these modifications, it is now possible to run the simulator across multiple processes with a large number of targets and target simulators. Lincoln Laboratory can reap immediate benefits from using this revised simulator in gaining the speedup of using multiple target simulator processes to distribute computation. It also readies the simulator for testing over a true distributed platform that spans multiple machines on a network.

6.2 Outstanding Issues

The revised simulation engine easily met the original metrics of success that were defined for the timing system and for securing critical sections. The main outstanding issue that remains with the simulation engine is simply performing additional tests to verify that no unexpected behaviors occur. This cannot be carried out over the course of a few days or even weeks, but will require significant usage to determine if any border cases arrive due to unforeseen issues.

6.3 Future Work

There are several logical next steps that can be taken to bring the simulator to an even higher level of fidelity. Topics of future work on the project are detailed below.

6.3.1 Middleware Configuration

In assuring that the simulation engine performs properly, the next logical step is to configure the simulator middleware to function over a truly distributed platform using RTCL. This was originally a component of this project, but was removed as the simulator was scaled down due to time constraints. Some progress has already been made in this direction, as the creator of RTCL was interviewed in order to gain help writing an RTCL configuration file for the network. A sample file was created that can be used for immediate testing over the network when time permits.

When executing over the network, RTCL uses a third-party software package called the “RTI Data Distribution Service.” The main work that remains is configuring RTI DDS within the RTCL XML configuration file and ensuring that buffer sizes and other parameters are within tolerable values. The actual XML configuration file is only 10–20 lines long, so the majority of the work should result from debugging as opposed to altering the configuration file.

6.3.2 Enhanced Models

The entirety of the work performed within this project was within the “engine” portion of the simulator. Now that the engine is stable, however, work should be put into the models to increase the fidelity of simulations. Currently, only point-source targets are being used with no environmental effects. Domain-specific knowledge should be leveraged within the models to increase the accuracy of the simulation and take advantage of the new speed and stability found within the engine.

6.4 Concluding Thoughts

The debugging and programming involved in revising the engine was a laborious task, but a necessary one to move forward with the DVERT simulation architecture. Great strides were made over the course of this project to identify issues when running the simulator, create a working timing system, as well as

secure all critical sections. On the conclusion of this project, the simulator is now stable and efficient enough to test truly distributed simulation over a series of networked machines.

References

1. Amdahl, G. M. “Validity of the single processor approach to achieving large scale computing capabilities.” In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey, April 18 - 20, 1967). AFIPS '67 (Spring). ACM, New York, NY, 483-485.
2. Apache Software Foundation. *Apache Subversion*. http://subversion.apache.org. 2010. Computer Software.
3. The Eclipse Foundation. *Eclipse IDE for C/C++ Developers*. http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/heliosr. 2010. Computer Software.
4. Fitch, Kevin. *CxxTest*. http://cxxtest.tigris.org. 2009. Computer Software.
5. Fujimoto, Richard M. *Parallel and Distributed Simulation Systems*. New York: John Wiley and Sons, 2000. Print.
6. Montgomery, Lyon, Scotta. “Distributed Virtual Environment for Radar Testing” October 14, 2010
7. MIT Lincoln Laboratory. *About the Lab*. http://www.ll.mit.edu/about/about.html. Copyright 2001-2010a. Accessed 19 October 2010.