3-21-2013

Estimating and Measuring Application Latency of Typical Distributed Interactive Simulation (DIS) - Based Simulation Architecture

Ryan L. Drinkwater

ESTIMATING AND MEASURING APPLICATION LATENCY OF TYPICAL DISTRIBUTED INTERACTIVE SIMULATION (DIS)-BASED ARCHITECTURE

THESIS

Ryan L. Drinkwater, Captain, USAF

AFIT-ENG-13-M-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-14

ESTIMATING AND MEASURING APPLICATION LATENCY OF TYPICAL DISTRIBUTED INTERACTIVE SIMULATION (DIS) - BASED ARCHITECTURE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Ryan L. Drinkwater, B.S.E.E.

Captain, USAF

March 2013

AFIT-ENG-13-M-14

ESTIMATING AND MEASURING APPLICATION LATENCY OF TYPICAL DISTRIBUTED INTERACTIVE SIMULATION (DIS) - BASED ARCHITECTURE

Ryan L. Drinkwater, B.S.E.E.
Captain, USAF

Approved:

Douglas D. Hodson	3/13/13
Dr. Douglas Hodson (Chairman)	Date
[illegible]	3/13/13
Dr. Raymond R. Hill (Member)	Date
[illegible]	3/13/13
Dr. Gilbert L. Peterson (Member)	Date

Abstract

One of the challenges in a distributed virtual environment stems from the requirement to simultaneously execute the simulations in real-time to support human interaction, in conjunction with maintaining a consistent view of the shared simulated environment. Maintaining a consistent set of simulation state data in the presence of network latency is difficult if individual data items are updated frequently. The principle application of Distributed Interactive Simulation (DIS)-based simulation environments has been in the domain of training where a consistent view or its correctness is often judged in subjective terms such as the simulation looking and feeling correct. New application areas for these systems are emerging in the analysis and test domains. For these domains, quantifying shared state consistency in terms of overall distributed application architecture is desirable. This research effort investigates and validates methods to calculate and measure the latency effects due to the multithreaded architecture of a real-time distributed simulation. These latencies may significantly affect the consistency of the simulation. An improved understanding is beneficial to the Air Force where real-time distributed simulations used for the purpose of analyzing the systems they simulate and the support of live test events.

Table of Contents

List of Figures

List of Tables

Page

List of Equations

ESTIMATING AND MEASURING APPLICATION LATENCY OF TYPICAL DISTRIBUTED INTERACTIVE SIMULATION (DIS) - BASED ARCHITECTURE

I. Introduction

General Issue

Real-time distributed simulations (RTDS) are used by the Department of Defense (DoD) to test systems and train personnel. One of the challenges with real-time distributed simulations is maintaining a tight consistency or small differences among all simulated views. Consistency is a measure of the difference among all simulated views. Consistency in a shared environment is affected negatively by numerous factors such as network latency, computational hardware, message ordering, and software architecture. A poor consistency in a real-time distributed simulation can lead to inaccuracies and lead to a poor portrayal of the simulated world. This chapter explores the problems associated with quantifying the consistency among real-time distributed simulations with a focus on latency due to software. Characterizing and quantifying simulation state consistency will lead to a better comprehension and prospective design of real-time distributed simulations.

An improved understanding of consistency within a simulation is beneficial to the Air Force where real-time distributed simulations are used for the purpose of analyzing the systems they simulate and supporting live test events. The U.S. DoD has used distributed simulations since the early 1990s as a cost-effective method for training its people in virtual simulations (Cavitt, Maly, & K.J, 1997). Within these training environments, people can learn without the real-world limitations such as safety, costs, training areas, or personnel (Cavitt, Maly, & K.J, 1997). In addition, simulations allow testing scenarios that would not otherwise be possible such as flying an experimental aircraft or testing the flight of a new weapon.

Problem Statement

Users and designers of real-time distributed systems need to understand the effects that software architecture have on consistency. When considering latency of the system, most research focuses on network latency exclusively. Although network latency does affect the consistency of the simulation, it is only one aspect and other sources should be examined. The architecture of the software may have significant impacts as well. Therefore, this research studies the impacts that software architecture has on the consistency of RTDS.

Interactive Simulations

The simulations of interest are often referred to by many different names. For example, Live-Virtual-Constructive simulations (), Distributed Virtual Environments (Zhou, Cai, Turner, & Zhao, 2003), networked Virtual Environment (Singhal & Zyda, 1999), RTDS all convey a similar type of simulation that share common characteristics. Regardless of the term used to describe these simulations, they all have commonalities. First, they are all distributed simulations, meaning they execute on separate physical machines often located at different geographic locations. This means that data must be shared across a network if the goal is to create a common shared virtual environment. Second, they are all real-time simulations that read, process and update state values with a specified amount of time. Most real-time "continuous" simulations executed on digital computers update the state that represents the virtual environment in discrete time steps, and these time steps must match a “wall clock” time or real world time. Because of this, real-time simulations must meet timing deadlines. In other words, the simulation must model the environment accurately at every discrete time step (Belanger, Venne, & Paquin, 2010) and perform this task in sync with the wall clock. Last, the simulators must be networked and share a single view of the modeled environment.

For a simulation to have a shared environment, it must have a shared sense of space, presence, and time (Singhal & Zyda, 1999), and it must do so by communicating state changes across a network. There are several standards for sharing data across the network. The IEEE Distributed Interactive Simulation (DIS) is one of the interoperability standards used as the basis for interconnecting individual real-time simulations. DIS defines the architecture for communication so that various types of simulations may be linked (Hofer & Loper, 1995). One advantage of DIS is that it allows interoperability so that systems built on different platforms, from different vendors, and/or from various services can all join in one simulation (Zalcman, 2004).

DIS-based simulations share a number of common characteristics. First, the individual simulations that compose the distributed simulation execute and operate autonomously and are responsible for maintaining the state of simulated entities-meaning there is no central server to control the entire simulation (Murray, 2010). Each application maintains its own view or *state* of the modeled environment. A *state* is “a complete description of a virtual entity at a single moment in time” (Churchill, Snowdon, & Munro, 2001). In addition, the term *object* must be defined. An *entity* or *object* is often used interchangeably, and represents an element of the synthetic environment that is created and controlled by a simulation application through the exchange of information

(Delaney, Ward, & McLoone, 2006). In this way, every simulator has its own virtual world that tries to synchronize with every other participant's virtual world. This is possible only by exchanging state data with participants.

Real-Time Distributed Simulation Consistency

One important aspect of RTDS is consistency. Consistency refers to the difference between all participants in RTDS. To maintain high levels of synchronization or tight consistency there are tradeoffs. To maintain a tight consistency among all participants, state data updates must occur often. However, if the state of a simulation is broadcast to other simulations too often, the network can become flooded with traffic that might result in degraded performance. If the simulation state is not broadcast often enough, the simulation might become poorly synchronized, resulting in errors. An ideal RTDS would be tightly synchronized which results in few errors present in the distributed system. This kind of synchronization is something real-time simulations strive to keep in an ideal system -- an absolute consistency. However, it is rarely possible. It takes time to transfer each participants view (state space) and for the receiving participants to process the state data and update their own environment. For this reason, there has been some effort in the field to strengthen the synchronization in real-time applications. These efforts focus on information exchange methods associated with algorithms (e.g. dead reckoning) that smooth the difference between the application's internal state with that of outside states (Zalcman, 2004). These methods help reduce network traffic by reducing the number of updates broadcast, while improving the consistency of views throughout the environment. There is however a lack of research in quantifying or characterizing the consistency of the real-time distributed simulations. In other words, the approach has always been to improve the consistency without knowing or quantifying the consistency itself. Furthermore, little research has explored the affect of application (i.e., simulation) architecture structure and its relationship to consistency measures.

Requirements for consistency may vary with each application. For some applications, the consistency may be relaxed in comparison to others. For example, a simulation may execute software on a host computer that models the virtual environment for the purpose of training or experimentation (Murray, 2010). In the training world, the consistency of a simulation is subjective, and the user is not directly concerned with the latency, but rather the "feel" of the simulation. For example, a user may not see a noticeable difference in views even if significant network delay is introduced, even though the simulation contains errors and inaccuracies. In fact, some argue that the absolute realism is not necessary for acceptable experiential uses of DIS (Dewar, Bankes, Hodges, Lucas, Saunders-Newton, & Vye, 1996). Instead, DIS needs only to give the users a good experience or proficiency in the training that they wish to receive from the simulation (Dewar, Bankes, Hodges, Lucas, Saunders-Newton, & Vye, 1996). In other words, if a user piloting an F-16 in real-time simulation releases a munition, the accuracy of the munition's impact during simulation is not as important as the user becoming proficient in the act-as the purpose for this simulation is for training. Although this thinking may apply to some training applications, there are new applications emerging in the analysis and test domains where understanding errors present in the system should be better understood. For these domains, characterizing the shared state consistency is important.

This research effort will give a better understanding of how the application architecture affects the consistency of RTDS. By quantifying a consistency of a real-time simulation, designers can properly address architectural parameters so the simulation meets the application requirements. The impact to the Air Force community could be significant as these systems are increasingly being used to support live test events and system analysis.

Research Goals

This research effort accomplishes several goals. RTDS are typically designed with concurrent executing threads that are responsible for updating the virtual environment, sampling state space data, moving data onto and away from the network, and updating the model or virtual environment. Typically these threads can be categorized as producers and/or consumers of the state data being exchanged. The rates at which each of these threads run can influence the consistency of the shared state data. This research quantifies the effects that each of these four thread rates have on the consistency of the state data. In addition, the results are expected to validate the research performed by Hodson (2009). The following questions are answered:

1. What effect do the four thread rates have on the consistency of RTDS?
2. Are the thread rate effects linear?
3. How do the metrics total latency, state data age, export time, and export error describe consistency?

Assumptions/Limitations

This research effort acts under some assumptions and limitations. The system to be tested is modeled with threads. Each thread performs its function within a specified period of time. The timers used have a resolution of 1 millisecond (ms); therefore, the period of each thread is only accurate to 1 ms, and this must be reflected in the data analysis.

Methodology Preview

To evaluate the software architecture, the essential aspects of a distributed virtual simulation are emulated by building a skeletal application to be executed on a real computer. This program was written in C++ using the Qt software framework. The emulator models a single object with a fixed velocity that moves along a straight path. As the object moves, differences in position between simulations are measured. The emulation is multithreaded and each thread sets a timestamp on the state data allowing a direct measurement of its age as it moves through the system. This experiment tests and analyzes four factors: *producer model thread rate, producer sample thread rate*, *consumer receiving thread rate,* and *consumer model thread rate*. To evaluate the interaction between all of the factors, a full factorial design is used. There are four factors, each with 2 levels. An Analysis of Variance (ANOVA) on each metric is performed to assess the difference in mean ages of the state data. The data shows the effect that each thread rate has on the consistency of the state data. In addition, the validity of this model is compared against the analytic model developed by Hodson

(2009), which characterizes the performance of distributed simulations in terms of temporal consistency.

Summary

In summary, dynamic RTDS are inherently inconsistent. Because it takes time to communicate state data to all participants in a shared simulation, there will almost always be some inconsistency in a frequently changing dynamic environment. There are a number of factors, which contribute to a lack of consistency between participants in RTDS. This research effort will examine a representative software architecture which several adjustable factors and determine its affect on consistency measures. This includes quantifying and modeling the thread rate effects on measurements of consistency for RTDS to better understand and improve shared state consistency.

Chapter 2 examines the factors affecting consistency, and the related research efforts in characterizing the consistency of RTDS in detail. Chapter 3 explains the use of an emulator to collect data on the metrics of consistency, and explains the methodology in detail. Chapter 4 discusses the results obtained from an experimental design that uses the emulator to represent the essential attributes of a distributed simulation. Finally, Chapter 5 presents the conclusions from this research effort.

II. Literature Review

Chapter Overview

This chapter discusses consistency in real-time distributed interactive simulations. With a focus on the real-time distributed simulations (RTDS) that are used by the Department of Defense (DoD) that operate and communicate using the Distributed Interactive Protocol (DIS). The consistency of the shared state data for these simulations depends upon several factors, including network latency, intercommunication protocols, and software architecture. In particular, many simulations are multi-threaded applications organized using the model-view-controller design pattern. The thread rates of the software design play a significant role in determining the state data consistency and are the focus of this research. Furthermore, this research builds upon the relevant work of Hodson (2009), which analyzes temporal consistency of a RTDS. Finally, the experimental design and collection of data is previewed. To collect data, a RTDS is emulated by constructing an application that models the thread interactions. The emulator allows direct measurements of consistency. The results of this experiment aid in quantifying and modeling the relationship and role that each thread plays with respect to different measurements of consistency of the simulation.

Terminology

The terminology associated with describing the different views and the propagation of data through a simulation are complex. This section defines and expands upon terms used throughout this document.

- *System* - an object or collection of objects whose properties we wish to study (Fritzson, 2003).

- *Simulation* - the imitation of the operation of a real-world process or system over time (Banks, Carson, B.Nelson, & Nicol, 2001). This term is used to refer to a computer simulation in this document. This term needs further clarification as it may refer to a single computer program running on an individual machine or it may refer to the collective sense of the term where a simulation constitutes multiple computer programs connected through a network. *Simulation* may be used interchangeably in these cases.

- *Model* - a representation of the system; in this case the system is the software architecture of real-time distributed simulation.

- *Consistency* - the difference between the models of all participating simulation programs.

- *Networked Virtual Environment* (net-VE) - a software system in which multiple users interact with each other in real-time (Singhal & Zyda, 1999). Each user accesses his or her own workstation or console using it as an interface to the virtual environment (Singhal & Zyda, 1999).

- *Distributed Virtual Simulation* (DVS) - a collection of independent computer simulations that appears to its users as a single coherent system (Tanenbaum, 2007).

- *Real-World Clock* or *Wall Clock* - a clock in the real world and not within a simulation. In other words, if the simulation starts at 3:00 PM CST and ends at 3:05PM CST then the simulation was executed for five minutes according to a real-world clock. Simulation clocks are based on the internal clock of the computer they run on. The advancement of time within a simulation does not necessarily follow a real-world clock.

- *Real-Time Distributed Simulations* (RTDS) - simulations that are connected over a network and meet real-time deadlines. These simulations are referred to as "real-time" for two reasons. First, the simulation advances in discrete time steps that match a real-world clock and external systems connected to the simulation operate as though they are connected to a real system (Roy E. Crosbie, 2007). Following a real-world clock allows the simulation to seem consistent across multiple computers and gives the illusion that all users share the same space and time.

- *Modeled Virtual Environment* - a virtual world as modeled by the simulation. This term may refer to a represented environment or a single virtual environment. A single represented environment is executed on a workstation that is networked to other workstations with represented environments.

- *State Space* - the sum of all variables that make up the modeled virtual environment in a computer simulation.

- *Shared State Data* - information exchanged between networked simulations, allowing each simulation to model the virtual environment. This information typically contains more than just information of a single entity. A simulation's state data may contain the position of many entities, weather conditions, terrain data, and the time of day for example. In this type of dynamic simulated environment, this information changes with time, and therefore, needs to be shared by every connected node (Singhal & Zyda, 1999). Described another way state data contains snap shots of the modeled virtual environment from any one simulation.

Shared View

The main goal of a networked RTDS is to provide the user with a shared view of a virtual environment. Ensuring that this view is consistent with every other view is one of the major challenges for software engineers. Consistency is a concept often seen in video games which simulate a physical world where players see the same objects. The requirement is referred to as WYSIWIS, or "What You See Is What I See" (Ravindran, Sabbir, & Ravindran, 2008). If an object in the simulation does not match its position in another player's simulation then there is a loss of player cohesiveness or what is called "consistency." The concept of "absolute consistency" is discussed later. If all players see exactly the same copy of the game state at the same time, then the game is said to have absolute consistency (Ravindran, Sabbir, & Ravindran, 2008). The same is true for any RTDS. As this chapter discusses, consistency is affected by several different factors and is the primary focus of this research.

Defining Consistency

Consistency the differences in state space from one modeled environment to the next. As a user interacts with a simulation their virtual environment changes. The changes must be reflected in all participating virtual environments. This shared virtual environment can be viewed as a collection of producers and consumers. For example, if a thread is dedicated to broadcasting local simulation state data to other participating simulations within the distributed system, this can be characterized as a sampling of local state space data. From there, consumers of that state data will receive that data and update their own local state information. Each simulation then uses their state space to model and draw 3D graphical views to present the virtual environment on a monitor or projection system. The inherent problem with this situation is that by the time consumers receive new updates, the producer's modeled environment may have already changed the value just communicated, which results in differences among the shared simulated environments (i.e., an inconsistency develops).

In a dynamically changing simulated environment, state data will have always "aged" by the time it is consumed, where "aged" simply means time has passed since the data values have been updated. Understanding how consistency affects the quality of the simulation, and what can be tolerated is associated with the simulations intended purpose (i.e., requirements). How "old" can the data be and still be considered valid is an underlying question to be answered. For much of the literature on this subject, to answer the question the age of the state data is examined, and a couple definitions for consistency are defined: absolute and temporal consistency. For a moving entity (e) in a RTDS, $P_i^e(t)$ represents the position of the entity, where *t* represents the wall-clock time, and *i* represents the simulation site the object is at. If $P_1^e(t) = P_2^e(t) = \cdots P_n^e(t)$ for *n* number of simulation sites then the application has absolute consistency (Zhou, et al., 2003). Temporal consistency is different from absolute consistency in that the data doesn't have to be the same throughout the participating simulations, but they must be within a *validity interval* (θ_{VI}) . The definition for temporal consistency also refers to the "age" of the data. For a shared data object θ, an application is temporally consistent if $\theta_{TS} + \theta_{VI} \geq t$, where θ_{TS} is the creation timestamp and θ_{VI} is the validity interval (Hodson, 2009).

Tradeoffs for Consistency

It is possible to maintain absolute simulation consistenty, but the tradeoff is simulation responsiveness. Sandeep et al. (1999) present the idea of Consistency-Throughput Tradeoff. This concept refers to the tradeoff between ensuring absolute consistency as opposed to allowing the dynamic shared state to change frequently (Singhal & Zyda, 1999). The only way to ensure absolute consistency is to not allow the state to change until all connected users have the same updated and shared state. In order to ensure this level of consistency, updates may take a long time. These long updates may be problematic for a simulation modeling a system at a high frame rate. For instance, if the network delay between two users is 150 milliseconds (ms), then to confirm that both users have the same shared state requires two confirmations across the network. This confirmation will take at least 300ms, which for a fast moving entity is significant. In fact, many real-time simulations require frame times in the order of milliseconds, and for some applications frame time rates are less than 10 μs. Therefore, for a simulated environment to be consistent it must be modeled at a rate that meets the application’s requirements. For this reason, absolute consistency poses a problem for real-time distributed simulations attempting to model virtual environments at fast rates. On the other hand, if absolute consistency is relaxed, then the simulation is less accurate in representing the position of entities. This is acceptable for some simulations. For example, in a flight simulator it is more important to simulate a smooth motion rather than simulate the position accurately (Singhal & Zyda, 1999). Therefore, the tradeoff is that absolute consistency is not maintained, but the simulation states can change at a speed that more accurately models and matches a real world clock. This tradeoff poses a problem for real-time distributed systems that wish to have both fast updates and a high degree of accuracy.

Consistency Versus Accuracy

Accuracy is related to consistency but the terms are not the same. The accuracy of the simulation measures how realistic the simulation matches the real world system it models whereas consistency is the difference between simulation views. In other words, a simulation may be grossly inaccurate but consistent with every other participating simulation. The accuracy of a real-time simulation is based on the dynamic representation of the system (Belanger, et al., 2010). As discussed earlier, there is a tradeoff for accuracy in a simulation that has a high rate of change. Accuracy is also based on the amount of time that it takes to model the system (Belanger, et al., 2010). The difficulty is in producing and processing model information within a given discrete time step (Belanger, et al., 2010). Therefore, if the simulation is designed to model the virtual environment every 10ms, then the computer must perform the following in less than 10ms: read input and generate outputs, solve model equations, exchange results with other simulation nodes, and wait for the start of the next (Belanger, et al., 2010). If these tasks are not met within time, inaccuracies may occur.

Factors of Consistency

Network Latency

One way to keep a tight consistency among nodes is to frequently transmit updates. However, there is another tradeoff when considering frequent state data updates. First, more nodes means more broadcasting, which can flood the network. Depending on the physical characteristics of the network, this frequent communication may limit the number of connected nodes, unless the consistency requirements are relaxed. Second, the more state data that is broadcasted, the more data that needs to be processed, and much of this data may be extraneous information (Singhal & Zyda, 1999). For example, if an entity broadcasts it position, but it is on the other side of an obstruction and cannot be seen, then much of this state data is not needed (Singhal & Zyda, 1999). In addition, there is latency associated with transmitting state data updates to other nodes.

Another significant factor to poor consistency within a simulation is network latency or lag (Delaney, Ward, & McLoone, 2006). Network latency for the most part is unavoidable. Furthermore, network latency is increased when simulators are dislocated. New technologies such as fiber optics have helped reduce network latency, but lag will always be associated with network communication. Therefore, it is important to be able to characterize latency and determine an acceptable threshold.

When examining the latency of a network, one must examine three aspects: bit propagation, packet processing and packet propagation delays. Large propagation delays relates to the physical speed of the transmission and cannot be avoided (Delaney, et al., 2006). Packet propagation delay refers to the "time required for all bits in a packet to be transmitted across the network from source to destination node considering only the inter-node bandwidth" (Delaney, et al., 2006). This delay can be reduced by increasing the network bandwidth (Delaney, et al., 2006). Packet processing delay is the time associated with the processing of the bits of data as it leaves the source and arrives at the destination. This delay is by reducing the quantity of data on the network. Reducing the quantity of data is accomplished by improving routers (Delaney, et al., 2006). For both the packet processing and propagation delays, reducing the number of state updates would most likely help, and is one way to improve network latency.

Jitter is another area of concern when it comes to characterizing the latency of an RTDS. Jitter is referred to as the unpredictable variation in latency with time (Delaney et al., 2006). In fact, jitter has a greater impact on performance than does latency (Delaney et al., 2006). It has been found that both performance and user strategy are affected by latency (Delaney et al., 2006). Therefore, examining and characterizing the latency are important.

Message Ordering

Another source of poor consistency is due to message ordering. Tolk (2012) explains why consistency suffers when messages are received out of order. Either there is an external delay such as a network lag, or there is an internal latency that causes the messages to be received out of order. Figure 1 shows an example of messages which are received in the wrong order due to internal latency (Tolk, 2012). The figure shows three icons each representing a networked simulator to include the Army, Air Force, and Navy. Imagine a scenario where an army tank is attacked by Navy artillery while an Air Force unmanned air vehicle (UAV) is observing. The Navy simulator sends out its state data to the Army and Air Force simulators notifying them that it has fired on the Army's tank. The Army simulator acknowledges and updates its environment, resulting in its tank being destroyed and sends this information out to both the Air Force and Navy. Due to latency in the message passing in this scenario, the Air Force UAV may see the Army tank destroyed before it sees the Navy Artillery firing. This kind of problem is known as a "time anomaly," and is corrected by managing the order in which it receives messages.

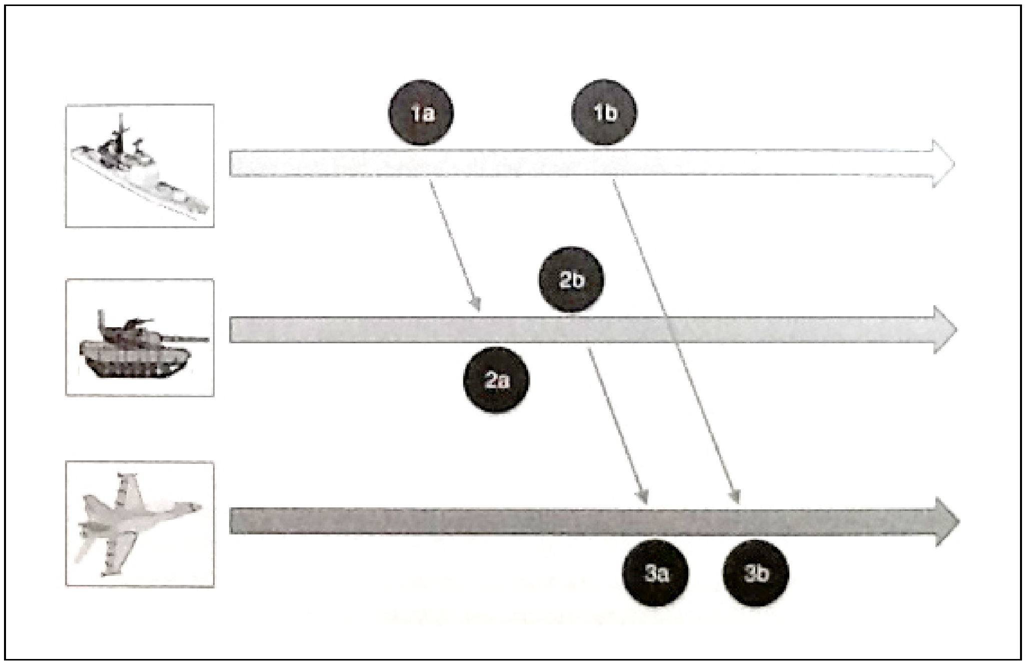


Figure 1: Time Anomaly (Tolk, 2012).

State consistency is also dependent on interacting model resolutions. Figure 2 shows Tolk's example of what is called "temporal inconsistency" (Tolk, 2012). If two simulations are connected, and the simulation on the left models an entity in one-minute intervals, while the simulation on the right models the entity at 15-minute intervals, a temporal inconsistency may exist (Tolk, 2012). The temporal inconsistency occurs because two truths exist, the entity as represented on the left, which is updated at a faster rate, and the entity on the right, which is updated at a much slower rate. One plausible solution in this situation is to give control of this entity to one of the simulations so both the left and right see a consistent view. However, this approach may result in a larger inconsistency (Tolk, 2012).

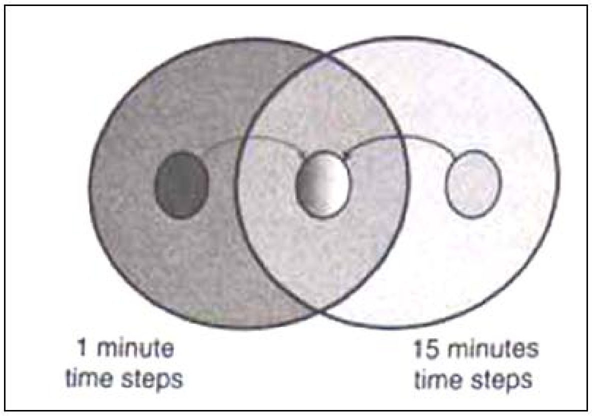


Figure 2: Temporal Inconsistency (Tolk, 2012).

Software Architecture

Another source of inconsistency is introduced by the architecture of the simulation software (Hodson, 2009). Most real-time simulations are created as multithreaded processes. That is, instead of producing, consuming, transmitting, and receiving state data sequentially, most software applications perform these processes using threads to take advantage of multiple core processors. Experience has shown that using a computer process with multiple threads to execute a task is better than a single sequential process (Bottazzi & Salati, 1991). A variant of the model-view-controller (MVC) design pattern is typically used to create these types of multithreaded simulations (Hodson, 2009). With this architecture, work is divided among the threads to draw graphics, model the environment and process network activity (Hodson, 2009). State data is touched by each of these threads from the moment it is produced to the moment it is consumed. Therefore, each of these threads potentially adds latency and affects the consistency of the simulation.

Relevant Research

Abstract models help in understanding the effects that software architecture has on state space consistency. Figure 3 shows an aggregate model for the software architecture used in Hodson’s analysis of state space consistency using a Petri net model (Hodson, 2009).

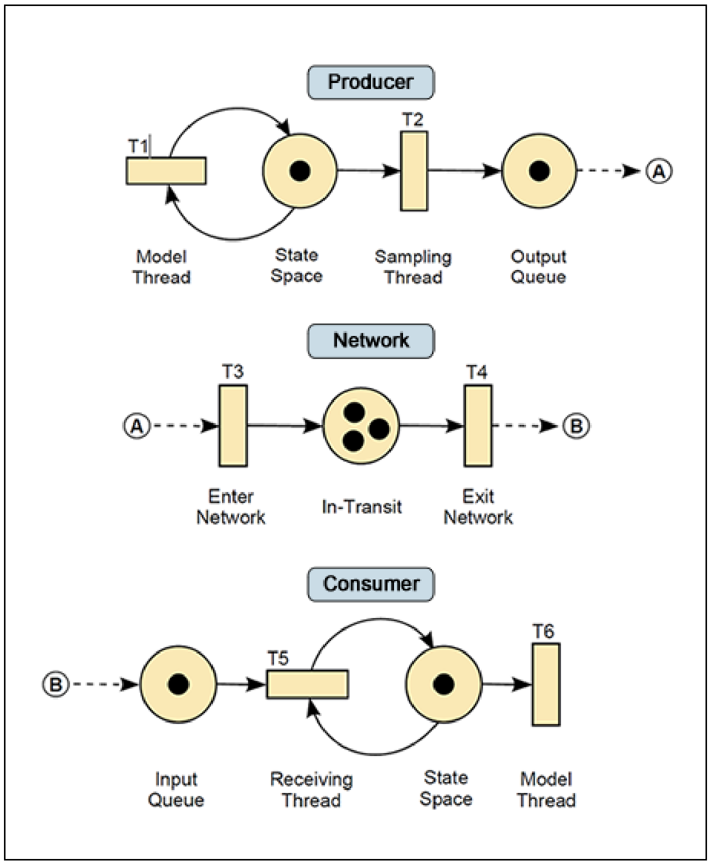


Figure 3: Producer, Network, and Consumer Models by Hodson (Hodson, 2009).

Each thread has a particular task and the threads are divided among the producer and the consumer tasks of the simulation. As one simulation produces state data and transmits this data through a network, another simulation acts as the consumer and processes the incoming state data. In Hodson's model (2009), the first thread or the model thread calculates and updates the environment local to the producer (i.e., updates the position of dynamic entities, etc). Snaps shots of this environment are synonymous to state space—a stored state of all modeled environmental variables. The next thread, the sampling thread, samples this state space and passes state data to other nodes (i.e., simulation) through a network infrastructure. These nodes act as consumers that contain a receiving thread to read the state data and update their own local state space. This is synonymous to receiving updates from all other nodes and then storing that information locally. The consumer model thread then uses the local state space to update its own modeled environment. Through this process, each node transmits and receives state data to create a shared virtual world in which all simulation instances or nodes view the same modeled environment.

Measuring Consistency

This research effort examines the effects that thread rates have on consistency. However, the term consistency can be associated with many different measures. For example, one possible measurement of consistency is the difference between the positions (i.e., data values) of an object modeled in two simulations, known as export error. Assuming this object is moving with constant velocity this may be a fair measurement of consistency in relation to thread rates. However, if the object is not moving then latencies really do not affect the consistency (because nothing is changing); thus, this metric does not always relay a true picture of the effects of thread rates. It is also complicated by the fact there are normally multiple objects modeled in simulations. To help characterize consistency to the fullest, this research effort examines four metrics that best describe consistency. These measurements are total latency, state data age, export time, and export error.

The system under test is a producer-consumer model that creates and consumes state data. The component under test is the thread architecture. The rates of the threads affect the consistency metrics. Therefore, the thread rates will be varied while examining the responses of the system. The methodology of this research effort will use an aggregate of Hodson's three models: Producer Model, Network Model, and Consumer Model (Hodson, 2009).

The System Under Test (SUT) includes the *producer component* and *consumer component* analogous to the producer and consumer models used by Hodson (2009). This system represents two networked simulations that are both producers and consumers of state data. The simulation architecture itself is an important aspect of this system. The architecture includes thread rates and state space of both the producer and consumer components. The thread rates affect how the state data propagate throughout the system until it is processed by the consumer model thread into the consumer's state space. The component under test (CUT) for this research is the *thread architecture component*. This research effort will evaluate the system using an emulator that allows direct measurements for each of the four metrics. It is hypothesized that the characterization of the threads will match and build upon the work of Hodson (2009). Models are created to predict the metrics of consistency. The next chapter discusses this methodology in detail.

Conclusion

This chapter has examined the shared state consistency problem of real-time distributed simulations. Due to network latency and the software architecture, there is potentially a difference in the shared state data. This difference between producer and consumer state data represents the consistency between them. Closely related research performed by Hodson (2009) used a Petri net model to simulate a consumer/producer system to characterize the age of the state data. This research effort will use actual threads to model and emulate a similar system and make direct measurements of the state data. The expected result of this experiment are models showing the relationship and the effects of each thread rate. The impact of this research is a better understanding of how application architecture affects the quality and/or consistency of the shared simulation state data. The next chapter discusses in detail the method by which data will be collected and analyzed.

III. Methodology

Chapter Overview

This chapter discusses the methods used to characterize the state data between real-time distributed simulations. First, a brief background is provided to define and discuss the problem. Second, the goals of this research effort and the objectives are presented. Next, the system, its services, parameters, metrics, and workload are introduced. Finally, a discussion of the evaluation technique, the experimental design, and the data analysis process is presented.

Problem Statement

One of the challenges and strategic goals with real-time distributed simulations is maintaining the consistency of the represented environments. Real-time distributed simulations (RTDS) must pass state data in order to correctly maintain the same represented environment. The state data used by each participating simulation provides a "view" of the represented environment. State space inconsistencies are the measured differences between each participant's view in a simulation. Simulations may have inconsistencies because the data cannot be transferred between simulations instantaneously, thus resulting participating simulations potentially using different state data. This research will examine the differences between these views. The problem is broken into a simpler model of producer and consumer called the State Space Consistency Model. This model is used to examine the differences of consistency between the consumer's view and the producer's view. The purpose of this research is to characterize the consistency of the state space affected by the software architecture of the simulation. There are several metrics used to measure consistency. This effort produces a model for each measurement of consistency and should follow and support the work performed by Hodson (2009).

Approach

Typically, the simulation is designed with a multithreaded architecture to allow the receiving, sending, and processing of state data to run concurrently. Figure 4 shows an aggregate model for the software architecture, which is based on Hodson's analysis of state space consistency using Petri net (Hodson, 2009).

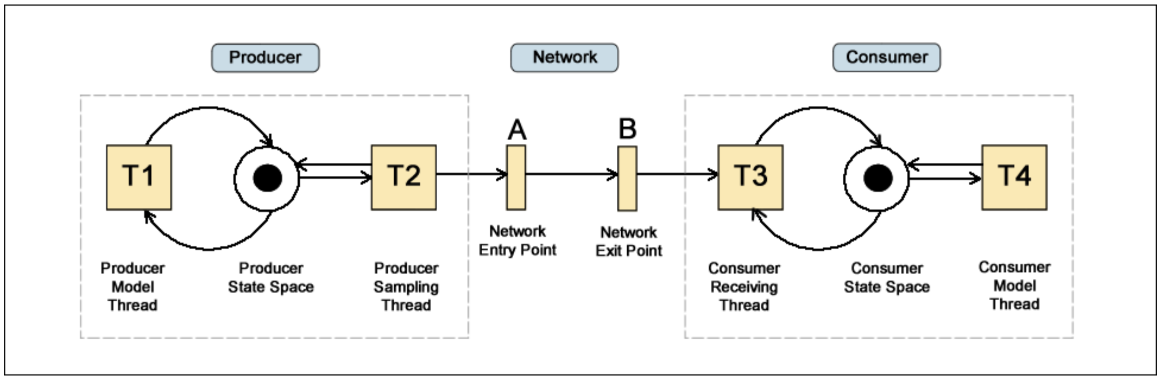


Figure 4: Multithreaded Architecture.

The state space consists of the variables that define the state of the modeled environment. Depending on the environment, whether it is modeling an aircraft or a missile, the variables change over time. For example, if this application were modeling an aircraft, the state space variables would likely represent the speed, altitude, and position of the aircraft. Other environment defining variables may include, weather, temperature, and terrain. The collection of all these variables define the state space of this application. Real-time simulations update their state as time progresses and matches time on a real-world clock.

Figure 4 shows the producer and a consumer state space. The producer model thread (T1) updates the producer state space; it updates the data associated with the systems it is simulating. The broadcasting of data through a network to other simulations is represented as a sampling of state space (T2). T2 transmits this state data over a network to a consumer. The consumer then incorporates the producer's shared state data. Finally, the consumer model thread (T4) updates the consumer state space based on the updated state. It is at T4 that the state variables from the producer are consumed (i.e., used) in the consumer's modeled environment.

Since real-time applications act as both producers and consumers of state data, it is easy to imagine that this process is bidirectional between two simulations as both receive each other's state data creating a shared view of a common modeled environment. In a networked system consisting of more than two simulations the same process takes place with a set of producers and consumers. The consistency of the shared modeled environment is ultimately dependent on the difference between the producer and consumer modeled environments. This consistency is more difficult to define and measure with multiple consumers and producers; thus, the analysis only considers a single producer and consumer. Even with only one consumer and one producer, the consistency of the modeled environment can be difficult to directly measure. One method for measuring the inconsistency involves examination of the differences of timestamps of state data from different points in the system. Typically, this is referred to as "age" because it measures the time of the state data from one point to the next in the system. However, age is a vague term as it could apply to several different metrics. In addition, just because the data has aged doesn't mean there exists inconsistencies in terms of data value differences. However, the system examined here is a dynamic system in which an entity is modeled with a constant rate of movement. Therefore, any time difference or aging in state data equates to an inconsistency in state data. There are several ways to define consistency, which are discussed in the performance metrics section.

System Boundaries

The System Under Test (SUT) includes the *producer component* and *consumer component*, analogous to the producer and consumer models described earlier. The SUT system represents two networked simulations described as one producer and one consumer. The simulation architecture itself is an important aspect in the SUT. The architecture includes thread rates and state space of both the producer and consumer components. The thread rates affect how frequently variables are updated and how data propagates through the system until the update occurs in the consumer's state space. The Component Under Test (CUT) is a single component within the SUT that will be varied. For this research the CUT is the *thread architecture component* as shown in Figure 5.

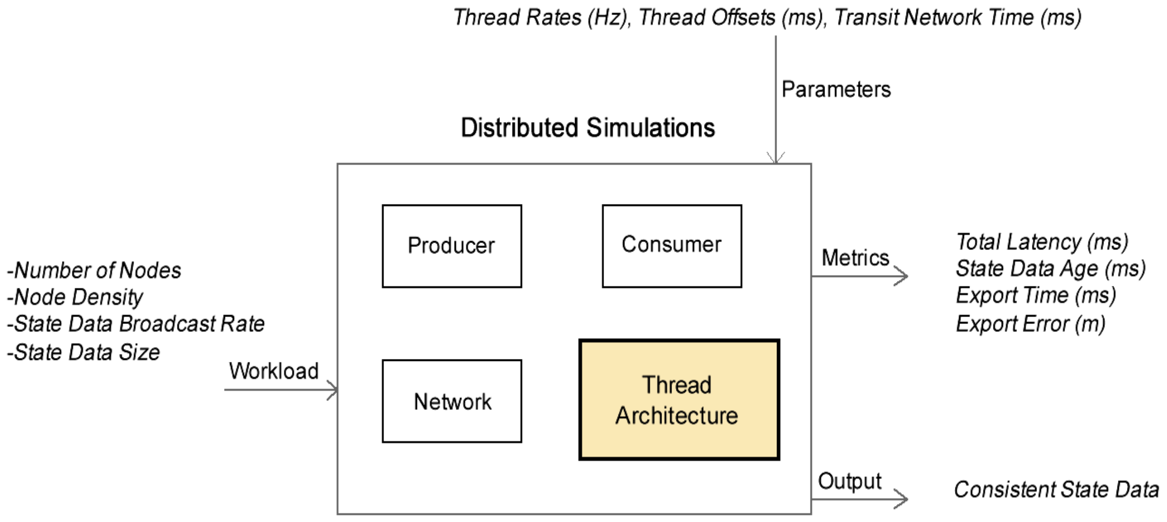


Figure 5: Real-Time Distributed Simulation System.

System Services

The service the SUT provides is a distributed interactive virtual environment. State data is used to create the simulation's model environment. Thus, every view or simulator connected should have a shared version of this model environment. The success of the service is determined by the application. If the application is for training, the consistency is relaxed allowing the environment to be responsive enough that the user does not notice inconsistencies within the virtual environment. However, for test and evaluation purposes, it is essential that the environment and consistency are high enough that the simulation models a real-world system to meet the objects of the experiment. The resulting state space's consistency is dependent upon the components within the SUT.

Workload

The workload for the system is associated with the amount of state data that is produced. The rate at which the model is updated by T1 and the sampling rate of T2 influence the workload. In addition, a higher node density and a higher number of nodes connected in the system results in more network communication, increasing the amount of data each node must process. The size and broadcast rate of state data also increases the workload of the system. The additional workload directly affects the level of consistency among every shared environment of each connected consumer/producer. Conversely, with fewer nodes, fewer data is processed by the system. In a typical real-world application of this system, the workload is application specific and widely varies.

Performance Metrics

There are many measures that can be used to define consistency, this research effort uses total latency, state data age, and export time shown in Figure 6. There is a fourth and separate metric that measures the difference in state data values called export error. Export error is measured by comparing the position of an object in the producer view versus the consumer view. For this particular experiment, export error is influenced by the rate of change associated with a single moving entity (i.e., object). The faster an object moves, the greater the export error. In addition, export error equals the export time multiplied by the velocity of the moving object.

The measurements are shown in Figure 6. The colored dots in the figure represent timestamps produced as each thread *touches* the state data. These timestamps act as a means to find several of the performance metrics. Note that in Figure 6, as drawn, there is a symmetric pattern for all the timestamps. This is rarely the case, but helps clearly describe the metrics in this context.

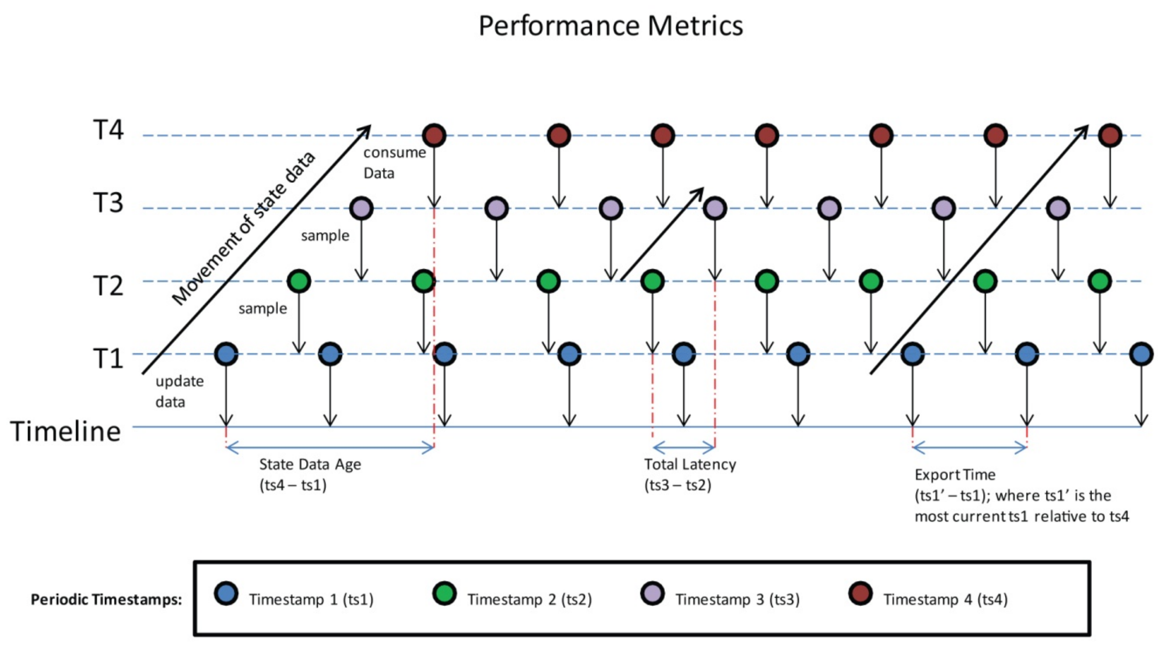


Figure 6: Performance Metrics

The following metrics are used to evaluate the performance of the distributed real-time emulation.

- *Total latency* (*ts3*-*ts2*) - the first measure is *total latency* of the state data. The difference in time from the point where state data is sampled at T2 (*ts2*) to the time it is received by the consumer at T3 (*ts3*) is defined as the *total latency*. Essentially, this is the time it takes to sample the producer's state data and transmit it across the network to the consumer (*ts3*-*ts2*), which then stores the data.

- *State Data age* (*ts4*-*ts1*) – the amount of time it takes state data to propagate through all four threads and how often the data is updated. This is from the time state data is updated at T1 (*ts1*), to the time it is consumed and used by T4 (*ts4*).

- *Export time* - neither total latency nor state data age account for the direct differences between producer and consumer state data. For this metric, the creation time of the state data used presently in the consumer and producer is compared. Figure 6 graphically shows this metric as ts1 subtracted from ts1', where ts1 is the creation timestamp of the state data currently used by T6, and ts1' is the creation timestamp of the state data currently used by T1.

- *Export error* - in addition to comparing timestamps of state data, it is possible to compare the values of state data variables between producer and consumer. The difference in one variable in a real-time simulation may not be a complete measure of consistency, because it is only one piece or small subset of variables. However, for this experiment, the emulator models a single entity and thus the object and its movement is the entire state space. Therefore, the difference in position of the object between producer and consumer is a good indication of the consistency. Therefore, the difference in an object’s position between producer and consumer is another measure of consistency, and is defined as the *export error*. This metric is recorded for every update received by T4.

Parameters

The parameters discussed below affect the performance of the real-time simulated environment.

- *Model Thread Rate* (T1) - the rate (hertz) at which the producer node updates its local state space by thread one.

- *Sample Thread Rate* (T2) - the rate (hertz) at which the producer's state space is sampled and broadcast into the network by thread two.

- *Receiving Thread Rate* (T3) - the rate (hertz) at which the consumer node receives data from the network and stores it in the consumer's state space by thread three.

- *Model Thread Rate* (T4) - the rate (hertz) at which the consumer node samples its state space by thread four.

- *Thread Offsets* - the random time offsets between executing threads, which captures the asynchronous dynamics of multi-threaded applications. The offset time (ms) between threads affects the performance metrics. The offset times are a modulus of each thread rate and are a result of initial startup dynamics associated with real multi-threaded applications.

Factors

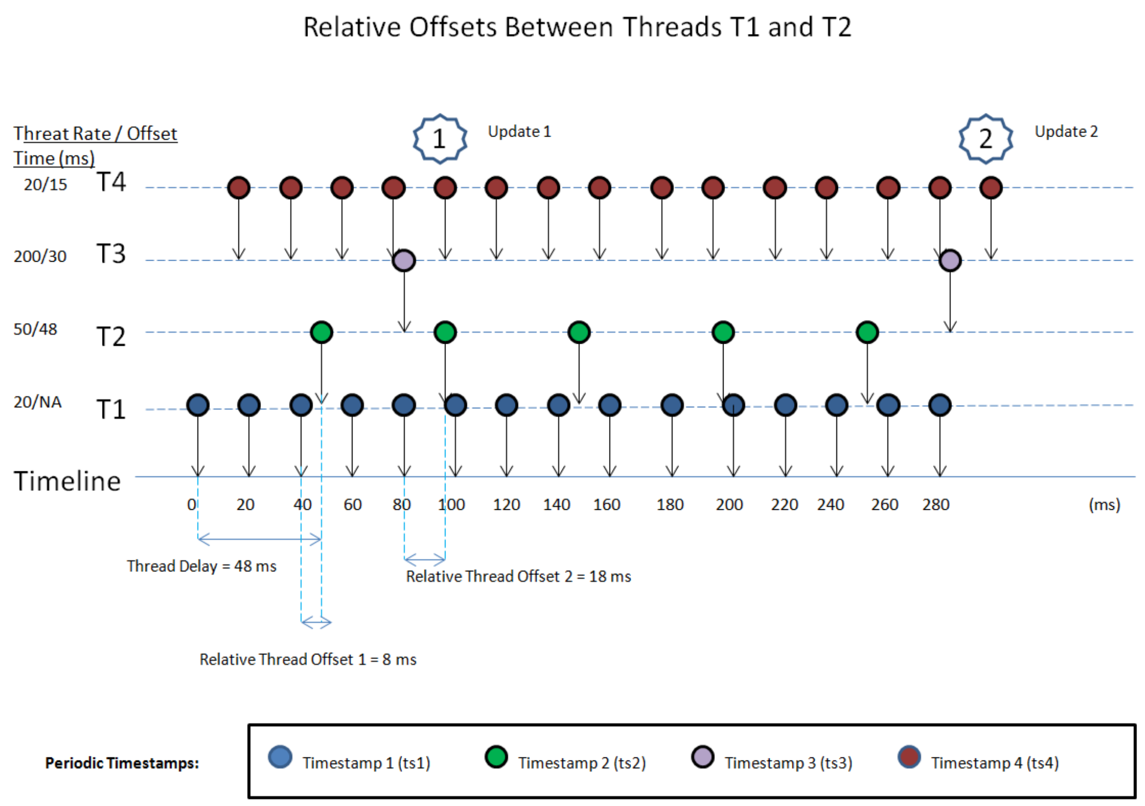
The following are the factors used in this research and each factors' corresponding levels. A summary of each factor level is found in Table 1.

- *Producer Model Thread Rate* (T1)
    - Small – 50 Hz
    - Large – 100 Hz
- *Producer Sample Thread Rate* (T2)
    - Small – 5 Hz
    - Large – 20 Hz
- *Network Latency* - a measureable unit of time delay experienced over a network. In a real geographically distributed simulation, network latency plays a significant role. However, for this experiment network latency is not a factor, and assumed to be negligible or near zero. In fact, for the emulator created, the data is placed on the network by the producer and then immediately received by the consumer.

- *Thread Delay* - a thread delay is the time added before the start of a thread to generate a relative thread offset between itself and the thread started before it. A thread delay is not a direct factor in this experiment, but is related to the thread rate. To properly emulate the initial conditions a real-time distributed simulation it is necessary to apply relative offsets between each thread on startup based on the respective thread rate. The offsets are accomplished by adding a random delay before the start of T2, T3, and T4. The delay for each thread is equal to a random number in milliseconds modulus the associated thread period (e.g., for a 10 ms thread period, the thread delay can b*e a random val*ue ranging from 1 to 10 ms).

- *Thread Offset* – the thread offset is the relative time between two threads and is not a factor in this experiment but results from thread delays. The thread delays are not necessarily equal to the relative thread offsets. This is because the threads are cyclic and thus the thread offsets are equal to the thread delay modulus the previously started thread's rate (e.g. for a thread delay of 48 ms for T2, where T1 has a period of 20 ms and T2 has a period of 50 ms, the first relative offset between T1 and T2 is 8 ms). Furthermore, thread offsets may vary with time. The difference between thread delays and thread offsets are shown in Figure 7. Figure 7 shows a thread delay of 48ms is added before the start of T2 relative to T1. Because T1 executes every 20 ms, the first relative thread offset between T1 and T2 is 8 ms. Notice that this thread offset varies with time. Since the next T2 execution is at 98 ms, the next relative offset between T1 and T2 is 18 ms.

- *Receiving Model Thread Rate* (T3)
    - Small – 5 Hz
    - Large – 20 Hz
- *Consumer Model Thread Rate* (T4)
    - Small – 50 Hz
    - Large – 100 Hz



Relative Offsets Between Threads T1 and T2
Threat Rate / Offset Time (ms)
Update 1
Update 2
1
2
20/15 T4
200/30 T3
50/48 T2
20/NA T1
Timeline
0 20 40 60 80 100 120 140 160 180 200 220 240 260 280 (ms)
Thread Delay = 48 ms
Relative Thread Offset 2 = 18 ms
Relative Thread Offset 1 = 8 ms
Periodic Timestamps:
Timestamp 1 (ts1)
Timestamp 2 (ts2)
Timestamp 3 (ts3)
Timestamp 4 (ts4)

Figure 7: Relative Offsets.

Evaluation Technique

To evaluate the system, the SUT was emulated by a software application to represent the essential architectural characteristics of a real-time distributed simulation. The emulator was built using C++ application based on the Qt framework. As most real-time distributed simulations use a multithread architecture, the emulator uses this same architecture and models the producer and consumer components as shown in Figure 7. In this experiment, the emulator models a single point object, which moves along a line at a fixed velocity in 2-D space. The path of the object uses the following physics equation for movement: $x = vt + x_o$, where x is the new position of the object, v is the velocity, t

is the time, and x_o is the initial position. Thread 1 (T1) is responsible for calculating and updating the object's position. Thread 2 (T2) samples the state space and transmits the position information to the consumer. The state data is then sent over the network and received by T3, which receives and stores the state data. The state data then moves to the final stage where T4 consumes or uses this data. There is difficulty in measuring the differences in timestamps between producer and consumer, because in a real-world simulations, each are executed on two physically separated machines, each with their own local clock. The local clocks, typically, cannot be perfectly synchronized, thus introducing error. To eliminate this obstacle, the emulator simulates the entire system-both producer and consumer-using one application, on a single machine, which enables the use of a single common time reference – this resolves a fundamental problem in measuring state differences without introducing the complexities of different time references. In addition, by using an emulator on a single machine, a shared memory block can be used to make comparisons (e.g., compare the position of the object as modeled by the producer and consumer at the same point in time).

Previous research by Hodson, looked at modeling the system using a Petri net model and analyzing the age of the state data from the creation and consumption from producer to consumer (Hodson, 2009). This research examined the direct experimentation of a real-time emulator with additional code added to record time stamps that allowed direct measurements in the differences of these timestamps. The resulting data was analyzed and a model was developed to estimate the performance of the system in regards to expected consistency, given thread factors. The results were compared and validated using the analytic model from Hodson’s work (Hodson, 2009). The trends were predicted to reveal the same general behavior of the system.

Experimental Design

Obtaining Measurements of Consistency

Figure 7 is an example of a producer/consumer network model and demonstrates three of the measurements of consistency. The example depicts the modeling of a simple object moving along a straight path with a constant velocity. The figure shows the state data consisting of the following state variables: position (x), initial/last position (xo), velocity (v), acceleration (a), timestamp 1 (ts1), timestamp 2 (ts2), timestamp 3 (ts3), and timestamp 4, (ts4). In Step 1, the producer model thread, T1, updates the producer’s model using the equation for motion. The object starts with an initial position of zero meters, a velocity of one meters/second and time equal to zero seconds. The state data variables are then updated at a regular frequency.

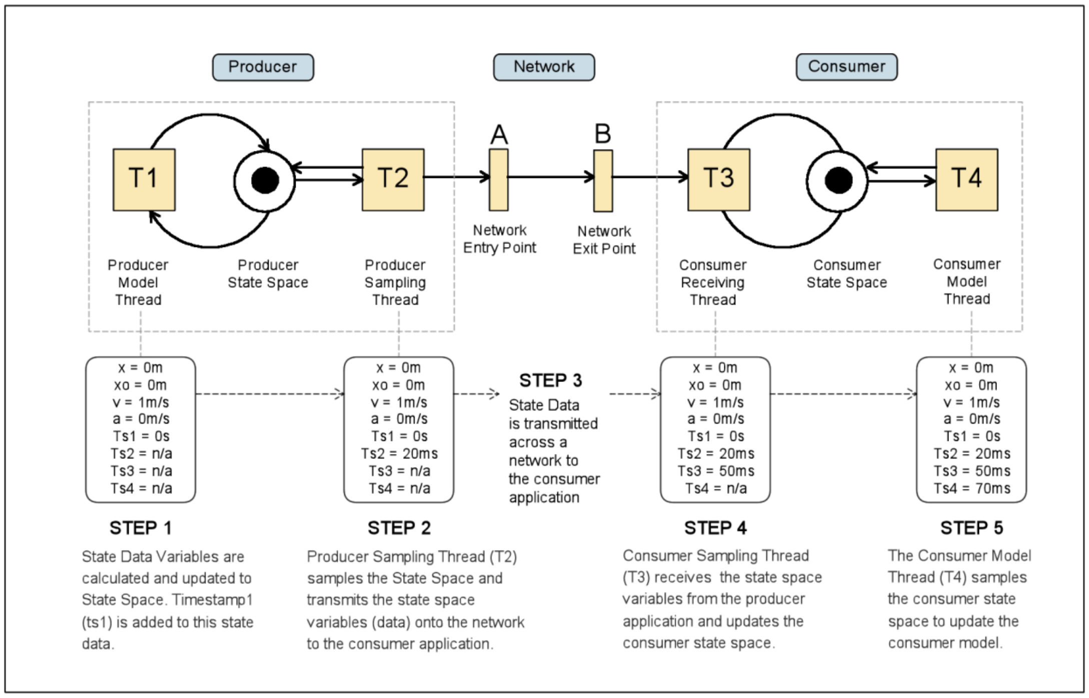

Producer
Network
Consumer
A
B
T1
T2
T3
T4
Network Entry Point
Network Exit Point
Producer Model Thread
Producer State Space
Producer Sampling Thread
Consumer Receiving Thread
Consumer State Space
Consumer Model Thread
x = 0m xo = 0m v = 1m/s a = 0m/s Ts1 = 0s Ts2 = n/a Ts3 = n/a Ts4 = n/a
x = 0m xo = 0m v = 1m/s a = 0m/s Ts1 = 0s Ts2 = 20ms Ts3 = n/a Ts4 = n/a
STEP 3 State Data is transmitted across a network to the consumer application
x = 0m xo = 0m v = 1m/s a = 0m/s Ts1 = 0s Ts2 = 20ms Ts3 = 50ms Ts4 = n/a
x = 0m xo = 0m v = 1m/s a = 0m/s Ts1 = 0s Ts2 = 20ms Ts3 = 50ms Ts4 = 70ms
STEP 1 State Data Variables are calculated and updated to State Space. Timestamp1 (ts1) is added to this state data.
STEP 2 Producer Sampling Thread (T2) samples the State Space and transmits the state space variables (data) onto the network to the consumer application.
STEP 4 Consumer Sampling Thread (T3) receives the state space variables from the producer application and updates the consumer state space.
STEP 5 The Consumer Model Thread (T4) samples the consumer state space to update the consumer model.

Figure 8: Multithreaded State Space Model Example.

Step 2 begins with T2 sampling the state data from the *producer state space*. T2 does not manipulate the data in any way and the only addition is T2’s own timestamp. T2 then transmits the new data across the network to the consumer, which takes place in Step 3. The consumer receiving thread, T3, receives the transmitted state data in Step 4, adds T3’s own timestamp to the state data, and then updates the *consumer state space*. Finally, in Step 5, the consumer model thread, T4, samples the state space, and adds its own timestamp. T4 then uses the current state data. The consumer model then shows the position of the object to be at 0m with a velocity of 1 m/s. The consumer model can be thought of as a second real-time distributed application; modeling the object in the producer model. Therefore, the modeled environment is of a simple object moving in a straight path that is shared by both the producer and consumer nodes.

Once the object modeled by the producer is represented in the consumer, an inconsistency in data between the models results from the time lapsed from when data was updated to when it is available for use. The total latency is found by subtracting *ts2* from *ts3*, which equals 30 ms for this example. In other words, it took 30 ms to transfer/deliver the state data to the consumer. The export time is found by subtracting *ts1* from *ts4*, which equals 70ms.

Measuring the time differences of state data is just one aspect of consistency. The difference between state variables in both producer and consumer views also is a measure of consistency. A difference in state data time does not mean the simulation has poor consistency. However, a difference in state data time means there is a potential that the state variables have changed. For example, if the object is not moving in the simulation then absolute consistency could be obtained even if there were significant delays in transmitting state data from producer to consumer. Therefore, the consistency depends also on the rate of movement of the objects being simulated. Following the example above, the data now used by the consumer indicates a position of 0 m. This is evident because it took in this example 70 ms for the state data to propagate from T1 to T4. Therefore, state data used to create the consumer model is 70 ms old. In addition, the position of the object has changed from the viewpoint of the producer, depending on the velocity of the object simulated. For example, if the velocity of the simulated object is 1 m/s and T1 updates the producer model every 10 ms, then after 70 ms the position of the object in the producer is 0.07 m while in the consumer it is 0 m. This error in the object's position is known as export error. All of these descriptors of state data - total latency, state data age, export time, and export error - represent metrics of consistency between the producer and consumer. Furthermore, the rate at which each thread updates contributes to all these measures of consistency. It is noteworthy to mention that the export error is dependent upon the movement of the object to include velocity and acceleration. For example, if the object had no velocity and was simply stationary, then the position would be the same in both the producer and consumer models; thus, they having an absolute consistency according to that measurement. If the object had a high velocity, with slow running thread rates, then the export error would be significant; thus, the consistency in this situation is often referred to as a loose coupling between producer and consumer.

Full Factorial Design

To evaluate the interaction between all the factors, a full factorial design is used. There are four factors, each with 2 levels as shown in Table 1.

Table 1: Factor Levels.

Factor	**T1 Rate**	**T2 Rate**	**T3 Rate**	**T4 Rate**
Level 1	50 Hz	5 Hz	5 Hz	50 Hz
Level 2	100 Hz	20 Hz	20 Hz	100 Hz

A full factorial design requires $2^4 = 16$ configurations. In addition, each run of the emulator will collect 100 data points. In other words, 100 datum packets will be propagate from T1 to T4. The averages for each of the metrics will be recorded. The emulator will be run by a script directing the emulator to shut down and restart in order to create new offset times and mitigate any unknown factors due to initial conditions. The script runs the emulator 1,000 times for each configuration, where a configuration is a set thread rate of T1, T2, T3, and T4. This means, for a 2^4 full factorial experiment, there will be 16 thread rate configurations. An average for each measure over the 1,000 runs at each configuration is recorded. Sufficient statistical basis for analysis is expected to be achieved with no more than five replications at 90% confidence. This results in 100,000 data points for each configuration, or 1,600,000 data points for each replication. Each iteration runs from the point where the first node produces the state data to the point where the second node consumes the data, and then repeating this process for 100 datum packets. Each producer/consumer node is modeled using C++ code and Qt framework running on Windows 7. Both an Intel Quad-Core 9550 and an Intel Six-Core i7-3930K processor were used.

Analysis

The data analysis supports the research goals to quantify the consistency affected by the architecture of a real-time distributed simulation. This consistency of the system is defined by difference of state data between producer and consumer. The timestamp difference of the state data is related to this potential difference. To assess the difference in mean timestamp differences of the state data, an Analysis of Variance (ANOVA) for each experiment is performed. It is presumed the thread rates have a significant role in determining the state consistency. In addition, the validity of this model is compared against the analytic model used by Hodson (Hodson, 2009).

Summary

Real-time simulations need to be accurate for use in emerging tests and analysis domains. Accuracy requires the simulated shared environment to be consistent. What is the effect that thread rates play in affecting consistency of a real-time distributed simulation? This goal of this research effort is to answer this question by emulating a real-time distributed simulation and collecting the data associated with the consistency of the state data as the data moves from the producing application to the consuming application. The experimental results from this research are expected to validate Hodson's (2009) analytic work on state space consistency. In addition, the models produced here are expected to be linear.

This chapter defined the methodology used in this research to evaluate and quantify the latency of state data due to the software architecture of a real-time distributed simulation. The system under test includes the producer, consumer, and a network. The service of this system is consistent state data. The metrics to measure the system include mean delay of the state data, throughput, and state data error. This experiment emulates a real-time distributed simulation and models the architecture of the system to analyze the effect thread rates have on the consistency of the state data. Finally, a full factorial experimental design is implemented and a method of analysis is provided.

IV. Analysis and Results

Chapter Overview

This chapter presents the results of the full factorial design using an emulator of a real-time distributed simulation (RTDS). This chapter also describes the data collecting process from the use of the emulator. After the data is collected, it is analyzed to determine how the thread rates affect the recorded metrics. This work includes the analysis of variance (ANOVA) results showing the significance of each thread rate, depicted several linear models. Given that four metrics were used to measure consistency, four linear models were developed. Thread rate significance varies among each metric, but overall it is evident that certain thread rates do significantly affect the responses. Furthermore, the results complement the research by Hodson (2009). Finally, results from this experiment aid in the understanding of consistency in RTDS used for testing and analysis purposes.

Process of Collecting Emulator Results

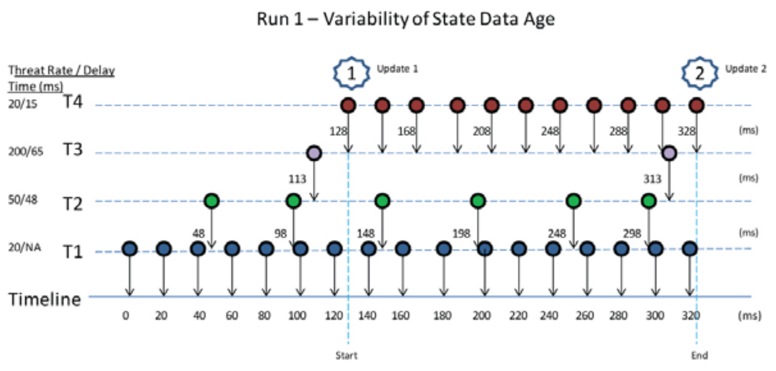
Overview of Emulator

The purpose of the emulator is to model a simple RTDS system. The emulator was created in C++ code and uses the Qt framework. Specifically, the emulator models four threads—*producer model thread* (T1), *producer sampling thread* (T2), *consumer receiving thread* (T3) and *consumer model thread* (T4). The emulator models a single point object moving along a straight path with a velocity of 1 m/s. Each thread executes at a specified rate and offset, with the exception of T1, which has no offset. As each thread becomes active it performs its designated function and state data moves through the system. Threads are started sequentially in the emulator with a random offset delay between each thread, which represents the asynchronous startup dynamics of multi-threaded applications. The four thread rates are used as factors, in 16 unique configurations, to constitute a 2^4 full factorial design. For each of the 16 configurations, 1000 *runs* were executed. Each run of the emulator collected 100 values for each metric of consistency: *export time* (ms), *total latency* (ms), *state data age* (ms), and *export error* (m). Finally, this process was replicated five times to ensure accurate representation of the system.

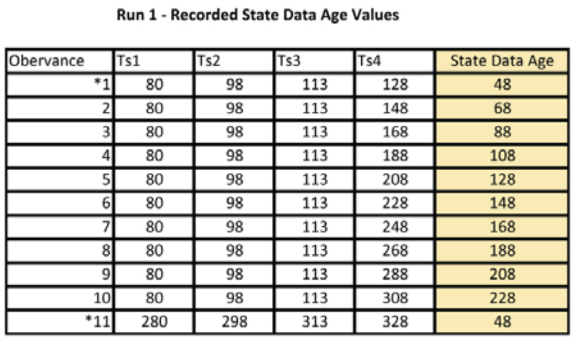
Experiment Configurations

Each *run* of the emulator captures a single configuration with fixed random delays between each thread. For each run, 100 observances are collected and averaged. Ideally, a single emulator run for each configuration would suffice; however, initial conditions and possibly other unknowns, vary the output. In addition, relative thread offsets, which are dependent on thread rates and delays, also vary the response significantly. Patterns in metric values are seen due to this variability and they depend on the metric, the thread configuration, and relative thread offsets. Figure 9 shows the variability of state data age within a run, where the state data age is at a minimum when T4 receives an update of state data. The state data age values increase until the next update. The thread rates and offsets determine the length of time between updates. In the example shown in Figure 9, the state data age increases through ten observances from 48 ms to 228 ms and then repeats this pattern.

For each run, the emulator stops and restarts with new random thread offsets. Metric values recorded between runs vary due to the latency added by these varying thread offsets even for the same thread rate configurations. The larger relative offset between threads adds more latency; thus, runs with larger relative thread offsets have greater times for metrics. The differences of a second emulator run are shown in Figure 10. The averages of the state data age values in Figure 10 are 117 ms compared to 138 ms in Figure 9. Due to this variability, many runs of a single thread rate configuration are executed. For this experiment, 1,000 runs per configuration were performed resulting in 1,000 averaged values per metric for each thread configuration.



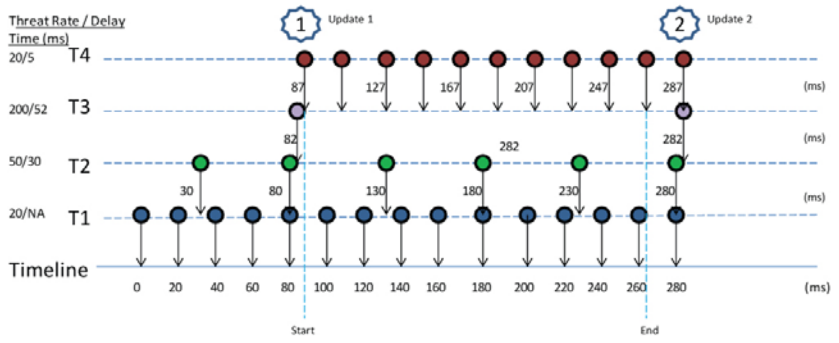
Run 1 – Variability of State Data Age
Threat Rate / Delay Time (ms)
1 Update 1
2 Update 2
20/15 T4
200/65 T3
50/48 T2
20/NA T1
Timeline
128 168 208 248 288 328 (ms)
113 313 (ms)
48 98 148 198 248 298 (ms)
0 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 (ms)
Start
End


Run 1 - Recorded State Data Age Values

Obervance	Ts1	Ts2	Ts3	Ts4	State Data Age
*1	80	98	113	128	48
2	80	98	113	148	68
3	80	98	113	168	88
4	80	98	113	188	108
5	80	98	113	208	128
6	80	98	113	228	148
7	80	98	113	248	168
8	80	98	113	268	188
9	80	98	113	288	208
10	80	98	113	308	228
*11	280	298	313	328	48

Figure 9: Run 1 Variability (State Data Age).

Run 2 – Variability of State Data Age



Run 2 - Recorded State Data Age Values

Obervance	Ts1	Ts2	Ts3	Ts4	State Data Age
*1	60	80	82	87	27
2	60	80	82	107	47
3	60	80	82	127	67
4	60	80	82	147	87
5	60	80	82	167	107
6	60	80	82	187	127
7	60	80	82	207	147
8	60	80	82	227	167
9	60	80	82	247	187
10	60	80	82	267	207
*11	260	280	282	287	27

Figure 10: Run 2 Variability (State Data Age).

Replications

A replication is a set of 16 configuration responses for each of the four metrics. These values are calculated from the average of the 1,000-recorded runs. For this thesis effort, five replications were performed to satisfy a 95 percent confidence level in the results. In addition, this experiment uses a significance level of 0.5. The calculated results for each metric over the entire experiment are captured in Table 2. The standard deviation between replications responses were about the same as the thread timing error of 1 ms.

Table 2: Experiment Responses (Averages of Five Replications).

Configuration	Thread Rates (Hz)	Thread Periods (ms)	SA (ms)	TL(ms)	ET (ms)	EE (m)
1	50/5/5/50	20/200/200/20	209.22	99.90	199.44	0.199
2	50/5/5/10	20/200/200/10	209.61	100.07	199.77	0.200
3	50/5/20/50	20/200/50/20	134.64	25.38	124.96	0.125
4	50/5/20/100	20/200/50/10	134.41	25.39	124.65	0.125
5	50/20/5/50	20/50/200/20	134.83	25.39	125.31	0.125
6	50/20/5/100	20/50/200/10	134.61	25.82	124.75	0.125
7	50/20/20/50	20/50/50/20	59.86	25.88	50.11	0.050
8	50/20/20/100	20/50/50/10	59.58	25.91	49.78	0.050
9	100/5/5/50	10/200/200/20	204.80	99.85	200.12	0.200
10	100/5/5/100	10/200/200/10	204.46	100.06	199.73	0.200
11	100/5/20/50	10/200/50/20	129.82	25.21	125.23	0.125
12	100/5/20/100	10/200/50/10	130.27	25.24	125.56	0.126
13	100/20/5/50	20/50/200/20	129.73	25.63	125.09	0.125
14	100/20/5/100	20/50/200/10	129.71	26.04	125.02	0.125
15	100/20/20/50	20/50/50/20	54.89	25.99	50.25	0.050
16	100/20/20/100	20/50/50/10	54.66	25.81	50.00	0.050

*State data age (SA), Total Latency (TL), Export Time (ET), and Export Error (EE)

Consistency Metric Results

Total Latency

Total latency is measured, from the time T2 samples state data, to the time T3 receives and stores the most recent state data. Once T2 transmits the data over the network it reaches a queue. T3 uses the most recent data stored on this queue. Once T3 has received data off the network queue, it does not receive data until a new update arrives from T2. This metric indicates the time it takes to move data new data from producer to consumer and is related to consistency of the system. The longer it takes to move state data from the producer to consumer the more inconsistent they potentially might be. Total latency, therefore, is a function of the thread architecture and network latency. However, for this experiment, network latency is negligible and close to 0 ms.

Figure 11 shows an example of an emulator response for total latency with a thread rate configuration of 50/5/5/50 Hz.

The variability of this metric is due to latency added by thread offsets between runs. In other words, within a single run, this metric is the same value. Once the emulator is restarted with new thread offsets and another run begins, a new value is observed, but it is the same value for all 100 observances. It is important to note that T3 receives only the most recent state data from the network queue. Therefore, total latency does not take into account "old" state data, and records only the time new state data propagates from T2 to T3. In other words, this metric will not increase if T3 checks the network again before an update has been received.

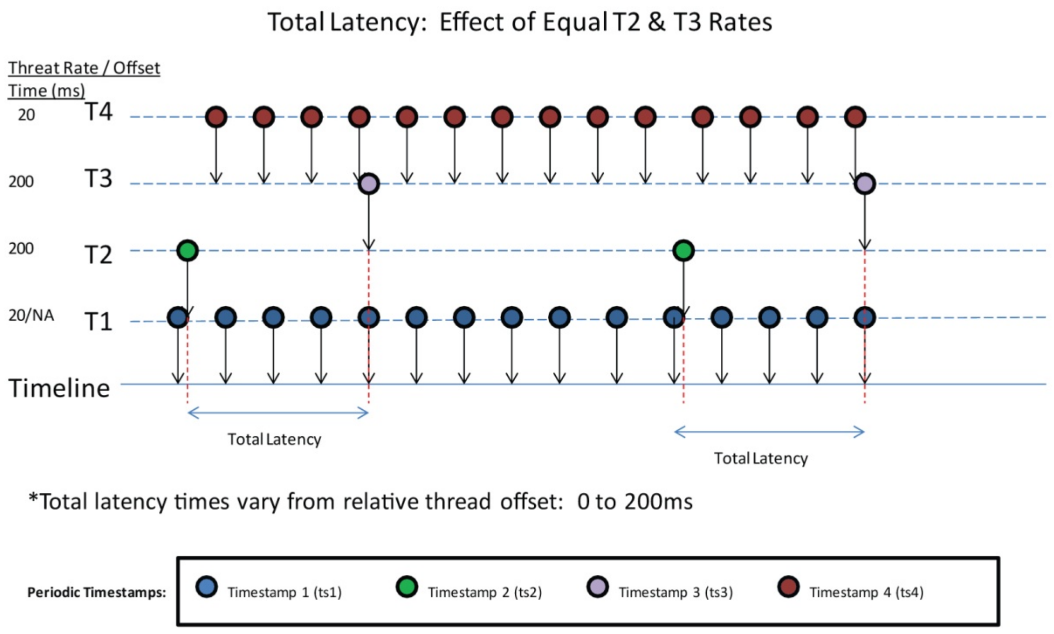


Figure 11: Total Latency Given Equal T2 & T3 Rates.

T1 rates have no effect on total latency because the measurement starts from the point T2 samples state data. The rate at which T1 updates this data has no impact on total latency. Similarly, T4 also has no effect since the end of this total latency measurement ends with T3 storing the data received from the network. Therefore, the rate at which T4 uses this stored data for calculations, etc, has no relationship to total latency. This leaves T2 and T3 factors. It is the relative offset between T2 and T3 that determines the overall latency or “freshness” of the delivered data plus any network latency. For this experiment network latency is practically zero. Therefore, total latency is dependent upon both thread rates. For this configuration in which T2 and T3 have the same rate (200 ms), total latency may vary anywhere from 0 to 200 ms (Figure 11). This is because the thread offsets associated with a 200ms period are from 0 to 200 ms. If the thread rates were increased, then the offsets are reduced and smaller total latency times result. For example, Figure 12 shows the impact of increasing the rate of T2. If T2 is reduced to 50 ms then the relative thread offset between T2 and T3 then can be anywhere from 0 to 50 ms for this configuration. The same reasoning applies if T2 and T3 rates are swapped.

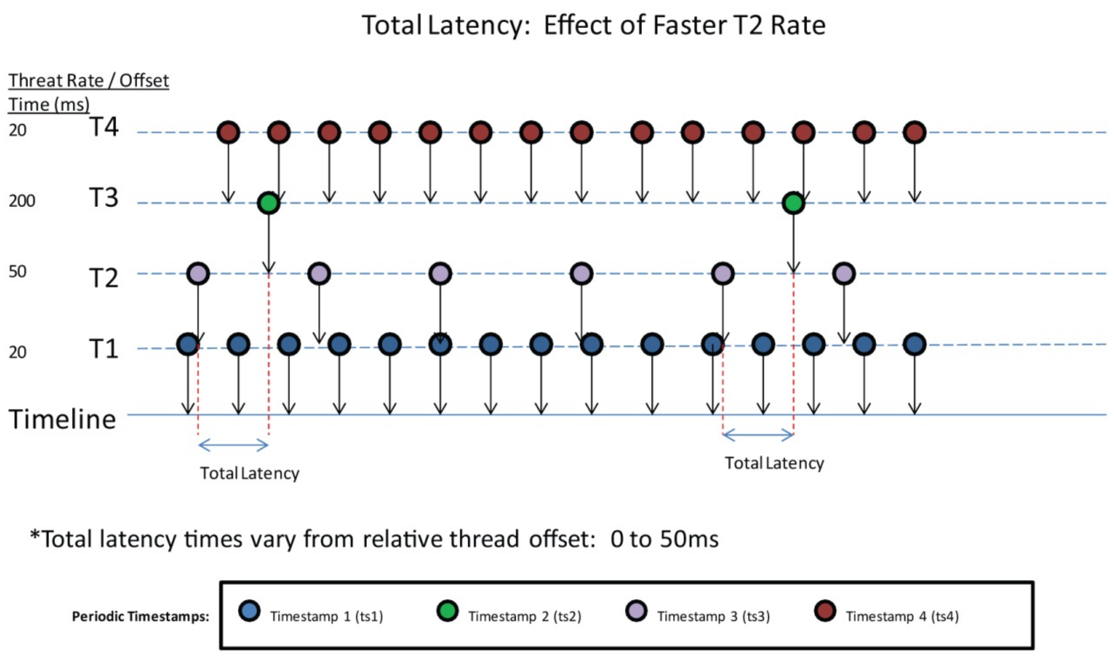


Figure 12: Total Latency With Faster T2 Rate.

When this is applied to the general case for all thread configurations, it is known that the lowest thread rate will have the largest influence on the result. This is because the threads are cyclic. Therefore, the fastest thread rate will determine the values. For example, if either of the threads has a 50 ms period, while the other is greater than 50 ms, then the relative offset between the threads will be values ranging from 0 to 50 ms. Therefore, the total latency will range from 0 to 50 ms. A general equation that describes the total latency based on thread rates and network latency is shown in equation 1. Since the equation is the average of responses, a factor of 1/2 is used. For example, if the smaller rate of T2 or T3 is 50 ms, then the average total latency would be 25 ms.

Average Total Latency :

$$TL = \frac{1}{2}\{Smaller\ of\ T2\ or\ T3\} + Network\ Latency \qquad (1)$$

To prove the total latency model correct, the output data is compared. As an example if the thread configuration of 20/200/200/20 ms, then the expected average total latency is 100 ms because the range of values vary from 1 to 200 ms. Likewise, for a thread configuration of 20/50/200/20 the total latency should be 25 ms. This is confirmed with the values presented in Table 2.

Average Total Latency indicates that both threads T2 and T3 play an equally significant role in determining how fast the most updated state data is delivered from producer to consumer. This is directly associated with the consistency of a simulation and should give insight to metrics discussed later in this paper.

State Data Age

State data age is a measure of consistency that indicates the time that has elapsed between the data the consumer is using and the time associated the current value as maintained by the producer. This metric is depicted in Figure 13 with a 20/200/200/20 ms thread period configuration. When state data is updated by T1, the data is “stamped” with the current time (i.e., timestamp) by the executing thread as part of the emulator. This data propagates through the system until it is used by T4, which then marks it with timestamp 4 (ts4). The difference between ts1 and ts4 is known as *state data age*. The state data age is based on how long it takes the data to propagate through the entire system and how often state data is used by T4. If no new update is used by T4 then the state data age gets larger. In this sense, this metric determines the “age” of the data.

The state data age varies within a run because this metric accounts for updates. In Figure 13, every dotted red line marks state data age measurement. Within a single run, this measurement grows until T4 receives *new* data. Because the thread rates of T4 to T3 are in a ratio of $\frac{1}{10}$, T4 will use the same state data 10 times before updated state data arrives. Therefore, state data age will increase by the value of T4's period 10 times. After T4 is updated the pattern will start again. Figure 13 captures the output of an emulator run, with the state data age increasing from 28 ms to 208 ms for every 10 observances. An average of 118 ms for this metric is then recorded to the cumulative output file.

The state data age also varies between runs due to random thread offsets set at the beginning of each run. A higher thread offset does necessarily add to state data age because the threads are cyclic. Instead, the relative offsets between

From this example it may appear that T4 may be impacting the average state data age, but it does not. For instance, if the period of the T4 rate were reduced by half, the average state data age for the same cycle of observances is about the same. In other words and averaging effect occurs, which negates the impact of the T4 rate. The influential factors in this example are T2 and T3, which not only help propagate the data through the system from producer to consumer, but also heavily impact the rate at which updates reach T4. These observances also apply to the general case for any thread configuration. The equation that describes the average state data age is shown below in equation. It is expected that β_4 and the intercept, I, will be approximately zero.

General State Data Age Model: $$SA = I + \beta_1(T1) + \beta_2(T2) + \beta_3(T3) + \beta_4(T4) \qquad (2)$$

An example of a single configuration run is shown in Figure 13 below. An entire replication records 1,000 of these runs for each of the 16 configurations. In addition, five replications of this experiment were performed. The average responses for all five replications are shown in Table 3. The largest standard deviation between all five replications for any of the configurations is 1.05 ms. An analysis of variance (ANOVA) was performed on the state data age responses (Table 4). The ANOVA indicates that in addition to T2 and T3 being significant, T1 also impacts state data age. In addition, a full linear model was created (Table 5), which includes all factors and interactions. Finally, with a significance level of 0.05, the t-test on coefficients in the model indicates that T1, T2 and T3 are significant, while other factors are not.

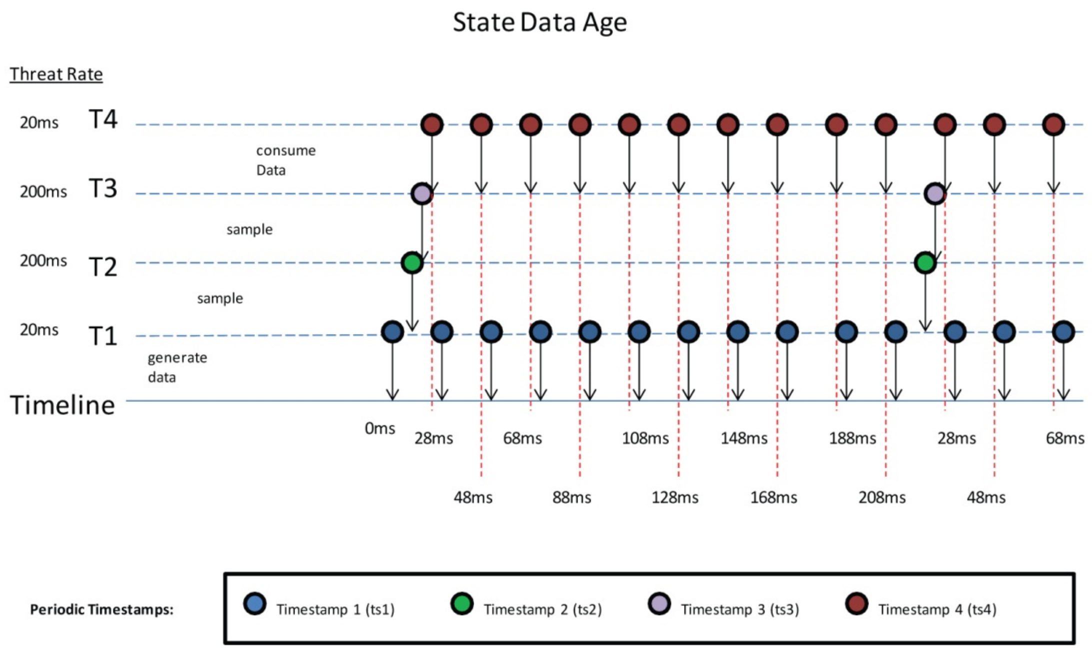


Figure 13: State Data Age Measurement.

Table 3: State Data Age.

Configuration	Thread Rates (Hz)	R1 (ms)	R2 (ms)	R3 (ms)	R4 (ms)	R5 (ms)
1	50/5/5/50	208.70	210.16	208.77	209.97	208.51
2	50/5/5/10	209.42	209.71	209.16	210.92	208.86
3	50/5/20/50	134.16	135.33	134.68	135.03	134.02
4	50/5/20/100	133.81	135.14	134.44	134.76	133.89
5	50/20/5/50	134.82	134.74	134.81	135.96	133.80
6	50/20/5/100	133.62	135.91	133.73	135.28	134.51
7	50/20/20/50	59.62	60.14	59.63	60.36	59.56
8	50/20/20/100	59.42	59.91	59.29	60.03	59.23
9	100/5/5/50	204.39	205.34	204.29	205.25	204.75
10	100/5/5/100	204.14	204.72	204.29	204.60	204.57
11	100/5/20/50	129.80	130.31	129.01	129.92	130.06
12	100/5/20/100	129.94	130.27	130.29	130.81	130.04
13	100/20/5/50	129.14	130.89	128.32	130.61	129.70
14	100/20/5/100	128.89	130.32	129.22	131.04	129.05
15	100/20/20/50	54.70	55.17	54.58	55.47	54.52
16	100/20/20/100	54.40	55.17	54.19	55.16	54.37

*R1-R5 = Replications 1 through 5

Table 4: State Data Age ANOVA.

Factor	DOF	SS	MS	F Value	Pr(>F)
T1	1	461	461	1.10E+03	<2e-16
T2	1	112271	112271	2.67E+05	<2e-16
T3	1	112069	112069	2.67E+05	<2e-16
T4	1	0	0	1.82E-01	0.6715
T1:T2	1	1	1	1.38E+00	0.2445
T1:T3	1	0	0	3.79E-01	0.5402
T2:T3	1	0	0	6.38E-01	0.4275
T1:T4	1	0	0	2.92E-02	0.8649
T2:T4	1	0	0	7.83E-01	0.3795
T3:T4	1	0	0	9.00E-03	0.9247
T1:T2:T3	1	0	0	1.85E-01	0.669
T1:T2:T4	1	0	0	6.26E-02	0.8033
T1:T3:T4	1	1	1	1.23E+00	0.2715
T2:T3:T4	1	0	0	1.40E-01	0.7094
T1:T2:T3:T4	1	1	1	1.79E+00	0.186
Residuals	64	27	0		

Table 5: State Data Age, Linear Model (Full).

Factor	**Estimate**	**Std. Error**	**t-value**	**Pr(>|t|)**
(Intercept)	-1.9700E+00	2.7360E+00	-7.2000E-01	4.7417E-01
T1	5.6740E-01	1.7310E-01	3.2780E+00	1.6900E-03
T2	5.2780E-01	1.8770E-02	2.8118E+01	<2E-16
T3	5.1250E-01	1.8770E-02	2.7304E+01	<2E-16
T4	1.0130E-01	1.7310E-01	5.8500E-01	5.6046E-01
T1:T2	-1.5100E-03	1.1870E-03	-1.2720E+00	2.0807E-01
T1:T3	-6.7320E-04	1.1870E-03	-5.6700E-01	5.7268E-01
T2:T3	-1.9610E-04	1.2880E-04	-1.5230E+00	1.3264E-01
T1:T4	-3.7420E-03	1.0950E-02	-3.4200E-01	7.3353E-01
T2:T4	-1.4370E-03	1.1870E-03	-1.2100E+00	2.3060E-01
T3:T4	-7.9300E-04	1.1870E-03	-6.6800E-01	5.0658E-01
T1:T2:T3	1.1420E-05	8.1440E-06	1.4020E+00	1.6577E-01
T1:T2:T4	7.6270E-05	7.5090E-05	1.0160E+00	3.1354E-01
T1:T3:T4	4.3490E-05	7.5090E-05	5.7900E-01	5.6451E-01
T2:T3:T4	1.1280E-05	8.1440E-06	1.3850E+00	1.7088E-01
T1:T2:T3:T4	-6.8690E-07	5.1510E-07	-1.3340E+00	1.8706E-01

Using just the significant factors a new model was created. This model is shown in Table 6 below.

Table 6: State Data Age, Linear Model (New).

Factor	**Estimate**	**Std. Error**	**t-value**	**Pr(>|t|)**
(Intercept)	0.1731	0.2757	0.6280	0.5320
T1	0.4803	0.0140	34.3600	<2e-16
T2	0.4995	0.0009	535.9810	<2e-16
T3	0.4990	0.0009	535.5100	<2e-16

Equation 3 shows the linear model for state data age. Analysis of this model shows that it is linear and there are no effects from the interactions of the threads. Therefore, each thread contributes a linear effect to the state data age. The adjusted R-squared for this model 0.9999 indicating an excellent linear model fit. In addition, R's global validation of regression assumptions indicates that skewness, kurtosis, link function, and heteroscedasticity assumptions are met by this model indicating a good linear fit. Figure 14 shows some of the model analysis plots. The Normal Probability plot indicates a good linear model within the 1 and -1 theoretical quintiles. In addition, the Histogram plot in Figure 14 indicates that the model has a near normal curve. A slight sinusoidal pattern emerges on the Plot of the Standardized Residuals, but this is to be expected because the influential factors, T2 and T3, have varying rates from fast to slow in the experiment.

State Data Age Model: $$SA = 0.48(T1) + 0.50(T2) + 0.50(T3) \tag{3}$$

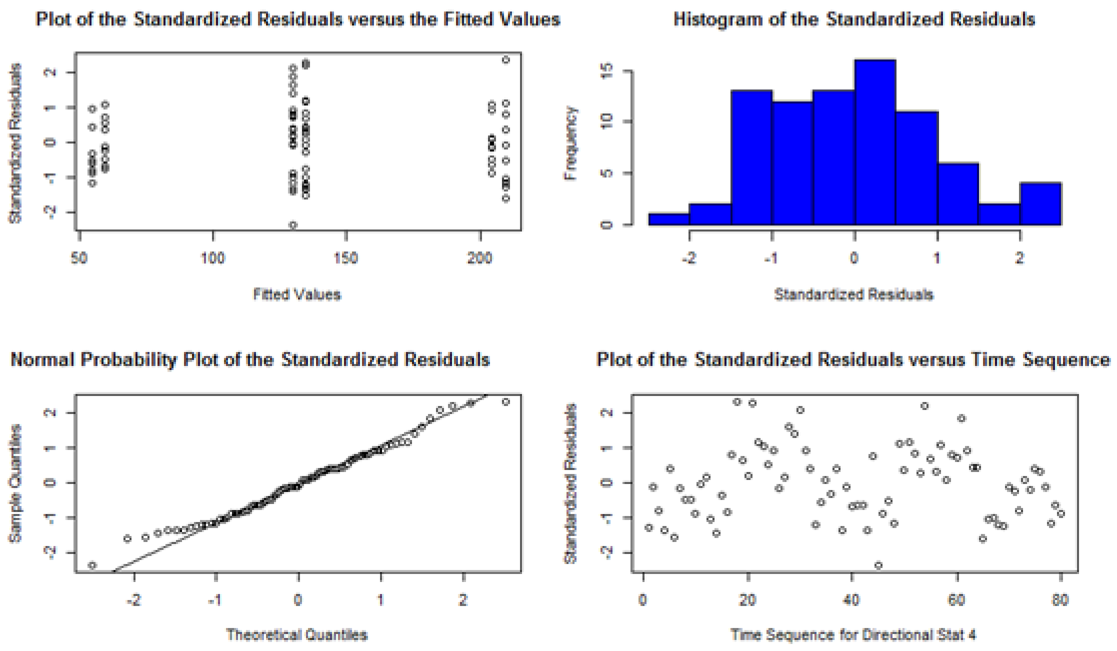


Figure 14: Model Analysis Plots (*State Data Age*).

The new model shows that threads rates T2, and T3 highly affect the state data age about equally. In addition, the model implies that T4 has no effect on the average state data age. Although T1 through the ANOVA is less significant than T2 and T3 it still impacts the state data age. Finally, the intercept for the new model is found to be negligible and well below the ± 1 ms error of the C++ timers and subsequently removed from the final model (Equation 3). This final model gives a great characterization of the export time and how the state data "ages" as it updated and used in the system.

Export Time

Export time measures the difference between the creation timestamp of data used at the consumer and of data being updated at the producer. *Export time* is an important metric for consistency because it gives a direct comparison of times of state data between the producer and consumer. The model is expected to be linear and is shown in equation 4 below. Based on the analysis of state data age, it is expected that β_4 and the intercept will be a negligible in this model. Furthermore, this model is expected to be linear.

General Export Time Model: $$ET = I + \beta_1(T1) + \beta_2(T2) + \beta_3(T3) + \beta_4(T4) \tag{4}$$

Five replications were performed during this experiment and the results are shown in Table 7. In addition, ANOVA was performed, and these results are shown in Table 8. The ANOVA results shown in Table 8 indicate that only T2 and T3 are significant factors. From the results of the ANOVA, a full model was created using all thread factors and interactions.

Table 7: Export Time Responses.

Config	Thread Rates (Hz)	R1 (ms)	R2 (ms)	R3 (ms)	R4 (ms)	R5 (ms)
1	50/5/5/50	198.87	200.14	198.59	200.57	199.05
2	50/5/5/10	199.65	201.00	199.17	199.75	199.29
3	50/5/20/50	124.59	125.19	124.31	125.83	124.88
4	50/5/20/100	124.05	125.00	124.22	125.19	124.79
5	50/20/5/50	125.09	126.48	123.94	125.44	125.60
6	50/20/5/100	123.78	125.39	124.53	126.09	123.96
7	50/20/20/50	50.24	50.60	49.85	50.14	49.70
8	50/20/20/100	49.73	50.05	49.45	49.96	49.71
9	100/5/5/50	199.75	200.51	200.18	200.49	199.67
10	100/5/5/100	199.55	199.77	199.90	199.94	199.50
11	100/5/20/50	125.17	125.41	125.54	125.71	124.34
12	100/5/20/100	125.26	125.94	125.48	125.46	125.65
13	100/20/5/50	124.52	125.84	125.16	126.30	123.63
14	100/20/5/100	124.20	126.18	124.53	125.58	124.60
15	100/20/20/50	50.04	50.80	49.89	50.53	50.00
16	100/20/20/100	49.73	50.32	49.77	50.53	49.64

*R1-R5 = Replications 1 through 5

Table 8: Export Time, ANOVA.

Factor	DOF	SS	MS	F Value	Pr(>F)
T1	1	2	2	3.81E+00	0.05535
T2	1	112188	112188	2.75E+05	<2.00E-16
T3	1	112012	112012	2.74E+05	<2.00E-16
T4	1	0	0	1.21E+00	0.27585
T1:T2	1	1	1	1.52E+00	0.22207
T1:T3	1	0	0	5.53E-01	0.45992
T2:T3	1	1	1	1.43E+00	0.23656
T1:T4	1	0	0	1.74E-01	0.67817
T2:T4	1	0	0	1.05E+00	0.31027
T3:T4	1	0	0	1.28E-02	0.91044
T1:T2:T3	1	0	0	4.11E-02	0.84001
T1:T2:T4	1	0	0	3.18E-01	0.57457
T1:T3:T4	1	0	0	6.62E-01	0.41888
T2:T3:T4	1	0	0	4.00E-04	0.98332
T1:T2:T3:T	1	1	1	2.38E+00	0.12761
Residuals	64	26	0		

The coefficients of the full model are shown in Table 9. Because only T2 and T3 were significant, a new model was created using only T2 and T3 rates while all other factors were dropped. To ensure the new model is sufficient, a drop-in-deviance test was performed. A drop-in-deviance test compares the full model against the new model to see if factors may be dropped without any significant loss to the model. This test indicated that all other factors may indeed be removed from the model.

Table 9: Export Time, Linear Model (Full).

Factor	Estimate	Std. Error	t-value	Pr(>|t|)
(Intercept)	-1.4920E+00	2.7000E+00	-5.5300E-01	5.8200E-01
T1	5.9330E-02	1.7080E-01	3.4700E-01	7.2900E-01
T2	5.2590E-01	1.8520E-02	2.8398E+01	<2e-16
T3	5.1450E-01	1.8520E-02	2.7781E+01	<2e-16
T4	1.0570E-01	1.7080E-01	6.1900E-01	5.3800E-01
T1:T2	-1.4480E-03	1.1710E-03	-1.2360E+00	2.2100E-01
T1:T3	-9.2840E-04	1.1710E-03	-7.9300E-01	4.3100E-01
T2:T3	-2.0020E-04	1.2710E-04	-1.5760E+00	1.2000E-01
T1:T4	-4.5010E-03	1.0800E-02	-4.1700E-01	6.7800E-01
T2:T4	-1.3500E-03	1.1710E-03	-1.1520E+00	2.5300E-01
T3:T4	-9.9090E-04	1.1710E-03	-8.4600E-01	4.0100E-01
T1:T2:T3	1.2280E-05	8.0350E-06	1.5290E+00	1.3100E-01
T1:T2:T4	7.6550E-05	7.4080E-05	1.0330E+00	3.0500E-01
T1:T3:T4	6.7050E-05	7.4080E-05	9.0500E-01	3.6900E-01
T2:T3:T4	1.1820E-05	8.0350E-06	1.4710E+00	1.4600E-01
T1:T2:T3:T4	-7.8450E-07	5.0820E-07	-1.5440E+00	1.2800E-01

The new export time model is created by dropping all factors but T2 and T3. This final model is shown in Equation 5 below.

Export Time Model: $$ET = 0.499(T2) + 0.499(T3) \tag{5}$$

Analysis of the export time model shows that it is linear and there are no significant effects from the interactions of the threads. Therefore, each thread contributes a linear effect to the state data age. The adjusted R-squared for this model is close to 1, indicating an excellent linear model fit. In addition, R’s global validation of regression assumptions indicates that skewness, kurtosis, link function, and heteroscedasticity assumptions are met by this model. Figure 15 shows some of the model analysis plots. The Normal Probability plot indicates a good linear model within the 1 and -1 theoretical quintiles. In addition, the Histogram plot in Figure 15 indicates that the model has close to a normal curve. A slight sinusoidal pattern emerges on the Plot of the Standardized Residuals, but this is to be expected because the influential factors, T2 and T3, vary in a cyclic manner over time.

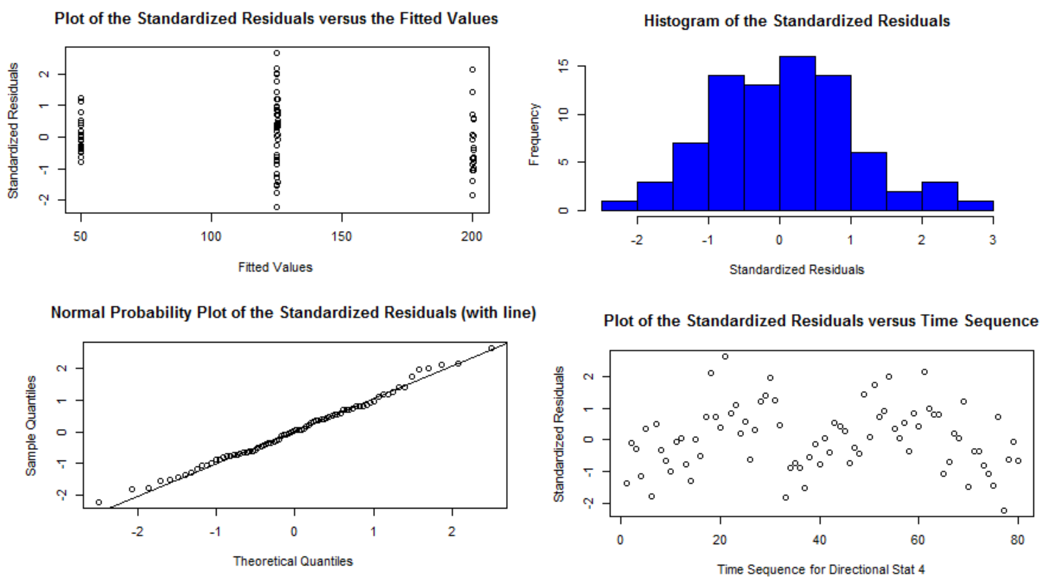


Figure 15: Model Analysis Plots (*Export Time*).

In this model, T1 does not have a significant effect on the responses. T1 is not influential because regardless of its rate, its contribution averages out to be the same. Figure 16 shows an example of a slower T1 rate. The export time measurements are represented by the red arrows in the lower half of the graph.

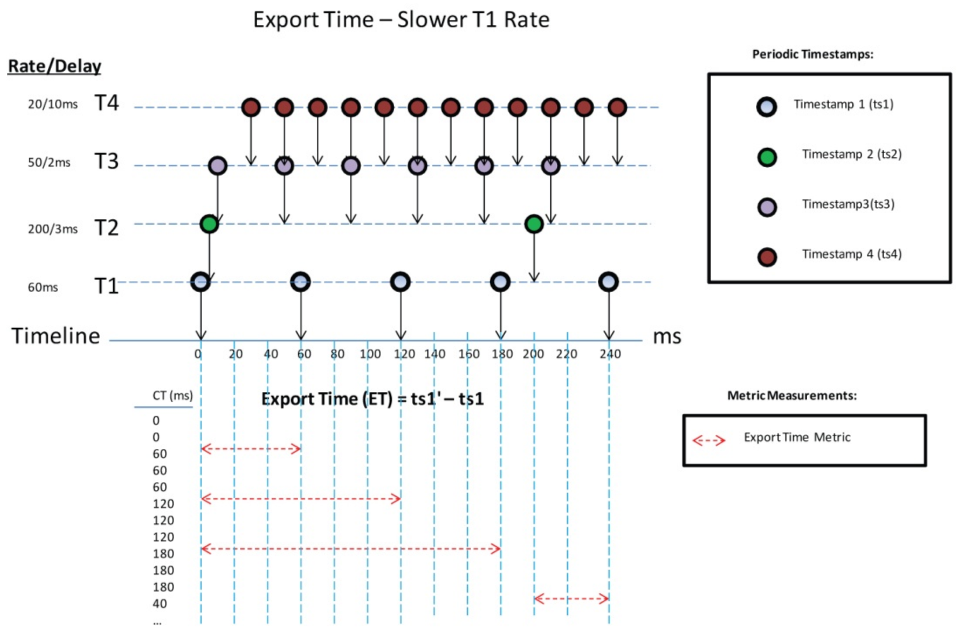


Figure 16: Export Time Measurement: Slower T1 Rate.

If T1 runs at a slower period of 60 ms then the *export times* will be 60, 120, 180, and 40 ms each repeated three times for the 12 observances by the consumer model thread. These responses average out to 100 ms. Now if T1 is run at a faster period of 20 ms as shown in Figure 17, then the *export times* for 12 observances the export times will be 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 40, and 60 ms. As with the slower T1 rate, the average of all of these responses is 100 ms. This example shows that over a number of observances over the same cycle the averages of the export times will be very close for any rate of T1. This averaging effect is depicted in Figure 18.

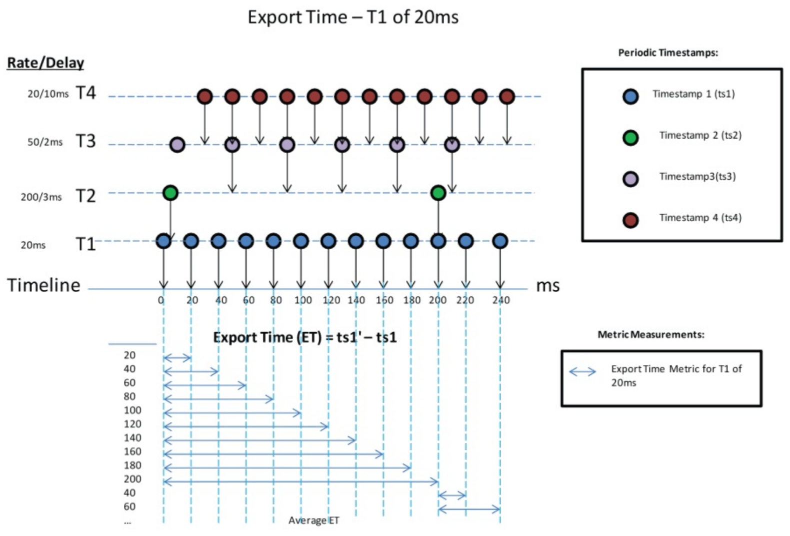


Figure 17: Export Time Measurement: Faster T1 Rate.

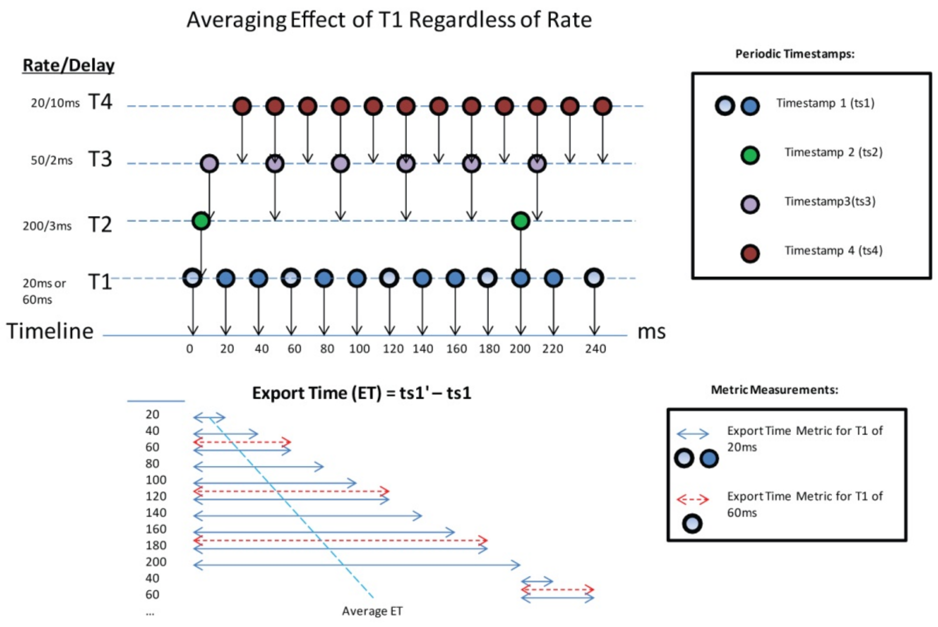


Figure 18: Export Time Measurement: T1 Averaging Effect.

What impacts the export times more significantly is the movement of state data from producer to consumer and this is dependent on T2 and T3. Therefore an increase in either the T2 or T3 rates results in lower export times. Figure 19 shows an example of this, with the blue asterisks above the T4 timestamps indicating updates of state data. If the T2 period is increased to from 200ms (as it was in the last example) to 50ms, the updates reach T4 faster and the consistency time measurements between T1 timestamps become more frequent; thus, the export times are less than if T2 had a longer period of 200 ms. In summary, faster T2 rates (smaller periods) result in smaller export times. The same explanation applies to T3—faster rates result in smaller export times.

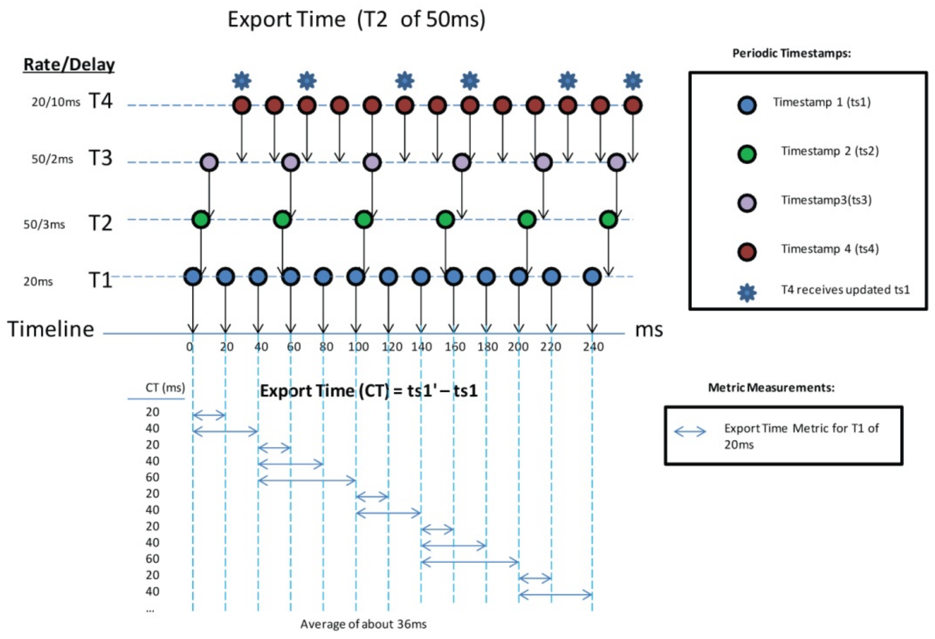


Figure 19: Export Time: T2 Effects.

Export Error

Export error is the measured difference of an object’s position at the producer and consumer. In this experiment, the velocity is a constant 1 m/s and the response is measured in meters. Since the *export error* is the difference in position from the producer versus the consumer, it is closely related to *export time*. The *export error* is equal to the *export time* converted to seconds (factor of 0.001) and multiplied by a velocity of 1 m/s. Since export times are recorded in milliseconds, the *export errors* gathered from the experiment should differ from *export times* by a factor of 0.001.

The resulting *export errors* were recorded from the same five emulator replications as the *export times*. The recorded responses in Table 10 show that *export errors* differ from *export times* by a factor of 0.001. As expected, the full model for export error shown in Table 11 are almost the same as the export time full model results. Likewise, the final linear model differs by a factor of 0.001 as shown in Equation 6. Note, that export errors are highly dependent on velocity and acceleration. For example, if the velocity were 100m/s then the linear model shown in equation 4 must be multiplied by a factor of 100. This research effort did not use modified parameters, but doing so in future research, may return interesting results.

Export Error Model: $$EE = 0.0005000(T2) + 0.0004996(T3) \tag{6}$$

Table 10: Export Error Responses.

Config	Thread Rates (Hz)	R1 (m)	R2 (m)	R3 (m)	R4 (m)	R5 (m)
1	50/5/5/50	0.20	0.20	0.20	0.20	0.20
2	50/5/5/10	0.20	0.20	0.20	0.20	0.20
3	50/5/20/50	0.12	0.13	0.12	0.13	0.12
4	50/5/20/100	0.12	0.13	0.12	0.13	0.12
5	50/20/5/50	0.13	0.13	0.12	0.13	0.13
6	50/20/5/100	0.12	0.13	0.12	0.13	0.12
7	50/20/20/50	0.05	0.05	0.05	0.05	0.05
8	50/20/20/100	0.05	0.05	0.05	0.05	0.05
9	100/5/5/50	0.20	0.20	0.20	0.20	0.20
10	100/5/5/100	0.20	0.20	0.20	0.20	0.20
11	100/5/20/50	0.13	0.13	0.13	0.13	0.12
12	100/5/20/100	0.13	0.13	0.13	0.13	0.13
13	100/20/5/50	0.12	0.13	0.13	0.13	0.12
14	100/20/5/100	0.12	0.13	0.12	0.13	0.12
15	100/20/20/50	0.05	0.05	0.05	0.05	0.05
16	100/20/20/100	0.05	0.05	0.05	0.05	0.05

*R1 - R5 = Replications 1 through 5

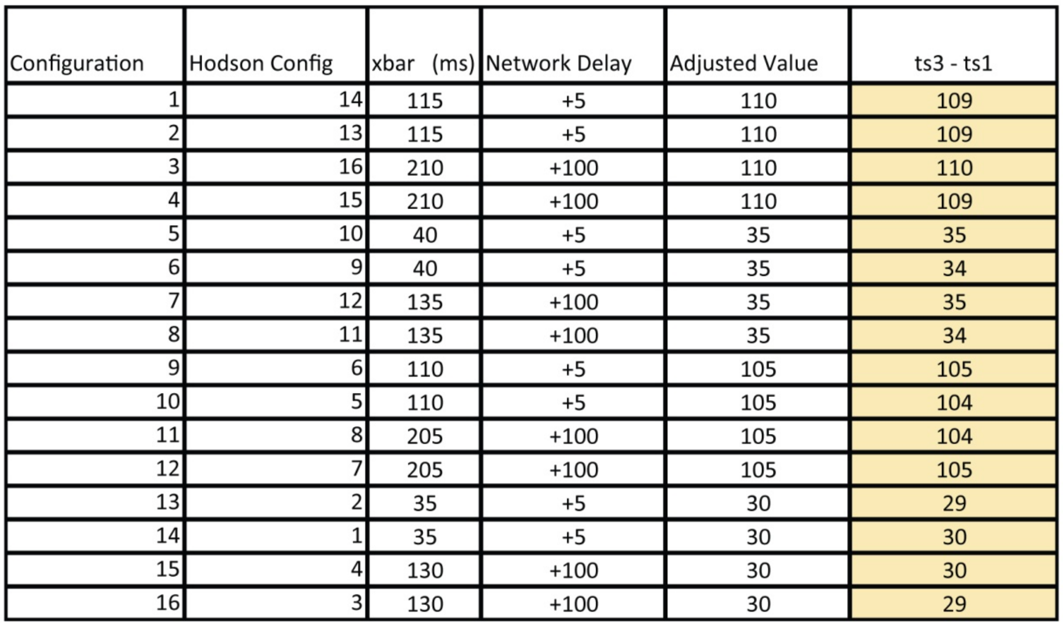
Table 11: Export Error, Linear Model (Full).

Factor	DOF	SS	MS	F Value	Pr(>F)
T1	1	0.000002	0.000002	3.8185	0.05506
T2	1	0.112187	0.112187	2.79E+05	< 2.00E-16
T3	1	0.112009	0.112009	278710	< 2.00E-16
T4	1	0	0	1.19E+00	0.27984
T1:T2	1	0.000001	0.000001	1.5154	0.22282
T1:T3	1	0	0	0.5592	0.45733
T2:T3	1	0.000001	0.000001	1.45E+00	0.23349
T1:T4	1	0	0	1.82E-01	0.67096
T2:T4	1	0	0	1.08E+00	0.30363
T3:T4	1	0	0	1.20E-02	0.91327
T1:T2:T3	1	0	0	4.04E-02	0.84129
T1:T2:T4	1	0	0	3.19E-01	0.57448
T1:T3:T4	1	0	0	6.87E-01	0.41024
T2:T3:T4	1	0	0	8.00E-04	0.97758
T1:T2:T3:T	1	0.000001	0.000001	2.41E+00	0.1256
Residuals	64	0.000026	0		

Comparison to Related Work

The experiment conducted in this research effort is an extension to the work performed by Hodson (2009). In Hodson’s work, a Petri Net was used to simulate thread architecture and interactions - this experiment obtained direct measurements by emulating the thread architecture of an RTDS. Because of the relationship to Hodson's work, it was expected to see related outcomes, and the models developed in both experiments should support the same characterization of the system. The metrics used in this research differ from Hodson’s. Since Hodson does not use thread three as a factor as data was instantaneously moved from the network into the consumer state space (simulating a blocked network thread), the model is similar to model used here but with T4 eliminated as a factor and a network delay added. Therefore, a new metric is measured using only the first three threads (ts3-ts1). This metric closely matches the metric used by Hodson (2009). In addition, Hodson adds a network latency factor, which by his own analysis is linear and additive to the state data ages. The ts3-ts1 values are then adjusted by adding the corresponding network delays. These adjusted values match very closely within ±1 ms of Hodson’s state data ages (2009). The comparison of these results are shown in Table 12.

Table 12: Comparison of Hodson's Results (Hodson, 2009).



Configuration	Hodson Config	xbar (ms)	Network Delay	Adjusted Value	ts3 - ts1
1	14	115	+5	110	109
2	13	115	+5	110	109
3	16	210	+100	110	110
4	15	210	+100	110	109
5	10	40	+5	35	35
6	9	40	+5	35	34
7	12	135	+100	35	35
8	11	135	+100	35	34
9	6	110	+5	105	105
10	5	110	+5	105	104
11	8	205	+100	105	104
12	7	205	+100	105	105
13	2	35	+5	30	29
14	1	35	+5	30	30
15	4	130	+100	30	30
16	3	130	+100	30	29

Hodson also concluded that T4 had no significance in determining the age of the state data (2009). The results found here concur that in all four metrics, T4 had no significant effect on the response. Furthermore, similar to Hodson (Hodson, 2009), the data recorded here indicates that T1, T2, and T3 are significant in determining the state data ages. What is interesting is that the export time is dependent only upon T2 and T3 rates. Thus, when considering consistency of data between producer and consumer only the rate of movement of data between them has a significant effect rather than the rate at which the models are updated.

Summary

The responses from all four metrics were used to create linear models and perform ANOVA to determine the effect each thread had on the results. The data and results have shown that the most significant threads in a Real-Time Distributed System are T1, T2, and T3. T4 had very little significance in all of the models presented and no significance in determining *export time*. In addition, the results, in comparison with results obtained from Hodson's (2009) work, verify expected similarities.

V. Conclusions and Recommendations

Chapter Overview

The purpose of this chapter is to present the conclusions of this research and explore recommendations for future research. Chapter one presented several research questions as to the effects of thread rates on consistency. These questions will be answered in this chapter and the answers will have an impact for future real-time distributed simulations (RTDS).

Conclusions of Research

The results of this research confirm that thread rates significantly affect the consistency of a real-time distributed simulation. For this research, four metrics were used to quantify consistency-export time, total latency, state data age, and export error. Among all of these metric responses, the results suggests that the consumer modeling thread (T4) has little effect. The rate at which the consumer models the virtual environment has little effect on consistency. What is more important, is that state data updates be available in a timely manner. For total latency, which is simply examining the time it takes the producer to transmit data to the consumer, T2 and T3 are the only significant factors. For state data age T1, T2, and T3 all are significant. Finally, for export time and export error the producer sampling thread (T2) and the consumer receiving thread (T3) have a significant effect on consistency because they are responsible for moving data from producer to consumer. This intuitively makes sense since consistency is more dependent on the rates at which updates are received at the consumer rather than the rate at which the producer and consumer model their respective environments. Therefore, T2 and T3 factors are significant because they heavily influence the rate at which data is transferred between producer and consumer and therefore influence their rate of updates at T4. Conversely, T1 and T4 model the environments of the producer and consumer respectively, but they have almost no affect on the export times and errors whether they update their models at slower or faster rate. In other words, they may update the models slowly, but as long as the state data from the producer is updated at the consumer at a fast rate, the simulation is consistent.

The overall results indicate that T2 and T3 are significant in keeping a strong consistency in the simulation. However, the consumer and producer modeling rates are less important. If these thread rates are slow then consistency of the models will be loose. If T2 and T3 threads rates are high, the state data is transferred more quickly and the consistency is high between producer and consumer. In addition, the rate of change of the entities simulated contribute to the consistency. For higher object velocities, higher export errors must be expected. Matching faster thread rates (T2, and T3) will help to improve this error, but there are limits and the speeds of these threads cannot be instantaneous. Thus, thread rates can be adjusted to help meet the requirements of the simulation. Furthermore, the models for each metric are linear, and support conclusions made by Hodson (2009). The data suggests that consistency depends on three factors—the rate at which entities in the producer model are moving, the rate of T2 and the rate of T3. Conversely, the rate at which the consumer model is updated has little effect.

The impact of this research is significant as software architecture significantly affects consistency – maybe more so than network latency. This research allows users to better understand, quantify and even estimate the effects of thread rates on consistency.

In addition, software should be designed with thread rates in mind to meet the goals of the simulation. Most users are aware of network latencies and know that such latencies can add 100 ms. However, the thread architecture with virtually no network latency may add up to on average about 200 ms. This is significant and should be accounted for in the testing and analysis. Finally, knowing average latencies of the software architecture plus network latency creates a baseline that can be established for the consistency of the simulation.

Recommendations for Future Research

There are several recommendations for future research. First, the emulator designed for this research was run using a single application. Future research would look to separate the emulator into two applications on a single machine to better represent a RTDS. Although, this would allow the emulator to still share the local system clock to make measurements, this would add complexity in timing of capturing the data as well as reading shared memory from each application.

Second, further research should also investigate network latency as an additional factor. Together, the architecture latency and the network latency should make up a more complete picture of the overall consistency of an RTDS. Constant network latencies should simply be additive, but introducing more complex transient network latencies would help characterize consistency under such conditions. An emulator measuring both architecture and network latency would provide a better representation of the system.

Examining more complex movements of multiple objects within the system might also provide some insight into consistency of RTDS. This research would tackle the question of defining consistency based on several differences within the system moving at different rates. This type of research could be complimented by examining dead reckoning algorithms, which seek to improve and reduce the communication between RTDS. By using an emulator that measures differences between applications, algorithms could be check for their effectiveness directly.

Finally, the applications could be separated to run on two machines. The emulator on the two applications could be adapted to send a system level ping from one computer to the next and then back again. Using synchronized clocks and recording timestamps, the time could be measured for the round trip that state data makes. This research would help give better insight to a system with two physically separated applications, which would better represent an RTDS system.

Summary

RTDS are used in a variety of applications-from training to test and analysis. In the latter case, it is important to have a strong consistency among all participants, or at least an understanding of it. Knowing the latency introduced by the simulation is important to understand and predict. By using an emulator to take direct measurements of a system similar to a RTDS, models have been created to help characterize the consistency. These models help characterize the system and validate previous research conclusions. This results here have only explored a few research paths dealing with consistency, and future research is needed. The impact that this kind of research has on the Air Force and the RTDS community is significant and will help shape future simulations.

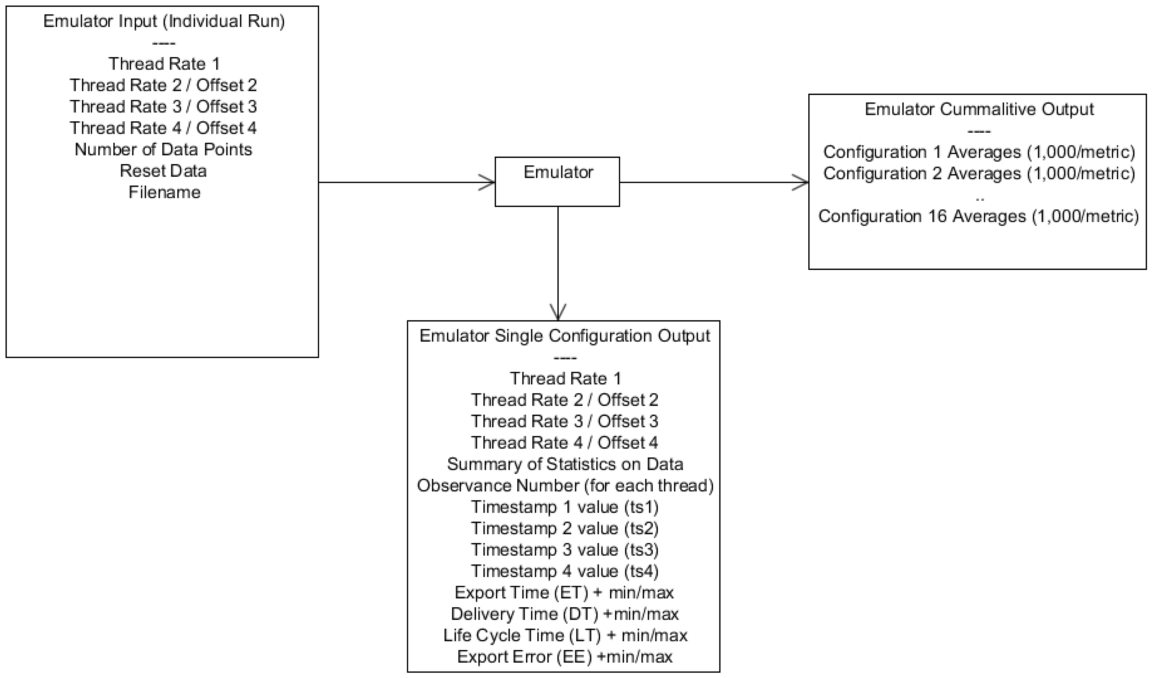
Appendix A

Appendix A Overview

The purpose of this section is to describe the emulator code used in this research effort to capture consistency metrics. The emulator models a simple real-time distributed simulation (RTDS) system by implementing four threads—producer model thread (T1), producer sampling thread (T2), consumer receiving thread (T3) and consumer model thread(T4). The emulator is able to adjust each thread rate in order to affect the response. In addition to each threat rate, offset delays are given to the emulator. The emulator allows direct measurement of the time at which data moves through the system. A description and UML diagrams are provided aid to explain the workings of the emulator code.

General Overview of Inputs and Outputs of the system

The emulator is given a set of parameters in which to run its test. For a single run, an emulator is given all four thread rates, thread offsets, number of data points to collect, reset data parameter, and a filename to save the data. This single run of the emulator starts a manager thread that starts each thread sequentially with the specified offset for threads two, three, and four. The thread offsets were initially calculated by the emulator but it was discovered the C++ *rand()* function did not accurately generate random numbers. For this reason, random delays were calculated outside the program. For this experiment 100 observances for *export time*, *export error*, and *state data age* are recorded before terminating the program. For total latency the observances must be recorded by T3, while all other metrics may be recorded by T4. These results are then written to a single configuration output file. This file contains header information such as the specified thread and offset rates, and statistics on the metrics. Each observance contains timestamps marked by each thread. As the emulator finishes, it writes the averages of all metrics to a cumulative output file. For each of the sixteen configurations 1,000 runs are executed producing 1,000 averaged values for each metric. Figure 20 depicts the emulator inputs and outputs.


Emulator Input (Individual Run)

Thread Rate 1
Thread Rate 2 / Offset 2
Thread Rate 3 / Offset 3
Thread Rate 4 / Offset 4
Number of Data Points
Reset Data
Filename
Emulator
Emulator Cummalitive Output

Configuration 1 Averages (1,000/metric)
Configuration 2 Averages (1,000/metric)
..
Configuration 16 Averages (1,000/metric)
Emulator Single Configuration Output

Thread Rate 1
Thread Rate 2 / Offset 2
Thread Rate 3 / Offset 3
Thread Rate 4 / Offset 4
Summary of Statistics on Data
Observance Number (for each thread)
Timestamp 1 value (ts1)
Timestamp 2 value (ts2)
Timestamp 3 value (ts3)
Timestamp 4 value (ts4)
Export Time (ET) + min/max
Delivery Time (DT) +min/max
Life Cycle Time (LT) + min/max
Export Error (EE) +min/max

Figure 20: Emulator Inputs and Outputs

Thread Architecture

The Qt framework simplifies the work of multithreading by using what are called Q-threads. Figure 22 shows a UML diagram describing the manager object and its associated thread objects. The manager's *readargs()* function is first called to store all the rate and delay parameters given to the emulator. Then the manager's *start()* function creates each thread and each associated process that will run. In order for the threads and the manager to communicate, Qt slots and signals are used as shown in Figure 21. In Qt, Q-threads and their associated processes are separate objects. The manager object moves these processes under each thread during run time and then calls the Q-thread object's *start()* function. Qt signals and slots are used to link each thread and its process. When a thread's process is complete it emits the *finished()* signal which is connected to the corresponding thread's *quit()* and *deletelater()* slot. Thus, the manager effectively is able to start and stop each thread. In order to control thread rates and delays. the Windows *timeGetTime()* function is used. This function retrieves the elapsed time since Windows was started. An initial start time is captured at the beginning of execution and then compared to the current retrievals of the system time. By this method an elapsed time is calculated thereby allowing threads to run at specified rates and introducing delays between thread start times. The Qt framework does offer Q-timers, but it was discovered that they are much less accurate (several milliseconds). In addition The windows *timeGetTime()* was found to be accurate within one millisecond.

Data Management

Thread one (T1) models a simple object moving in a straight path. The velocity is set to 1m/s and its initial position is zero meters. T1 calculates the new position of the this object according to its thread rate. If T1 has a rate of 50 Hz then every 20 ms it will recalculate the position of the object. It stores the position of the modeled object in a *mydata* object along with a timestamp, ts1. Thread 2 (T2) then samples the data from T1's *mydata* object. It reads all the variables and stores its own timestamp (ts2) in the *data* object. It then transmits this via UDP on the local network. Thread 3 (T3) reads the local network port for this data. It then stores the received data into the consumer's *mydata* object along with its own timestamp (ts3). Thread 4 (T4) then samples the *mydata* object and stores its own timestamp (ts4). The emulator then appends all received data into a running QString variable called *totaldata*. When enough observances are recorded, the process will quit and emit a signal to the thread manager object to save the data. The thread manager then gather all the data from the thread process, and write this to file. In addition, it will save metric averages to a cumulative output file. Finally, the program terminates.

Summary

The key to this experiment was having an emulator that could directly measure the time between each thread in the system. The emulator was created using C++ in a Qt framework. Many of the Qt libraries helped to make the program simpler such as QThreads. Overall, the emulator performed its function and the data recorded was used to characterize the system.

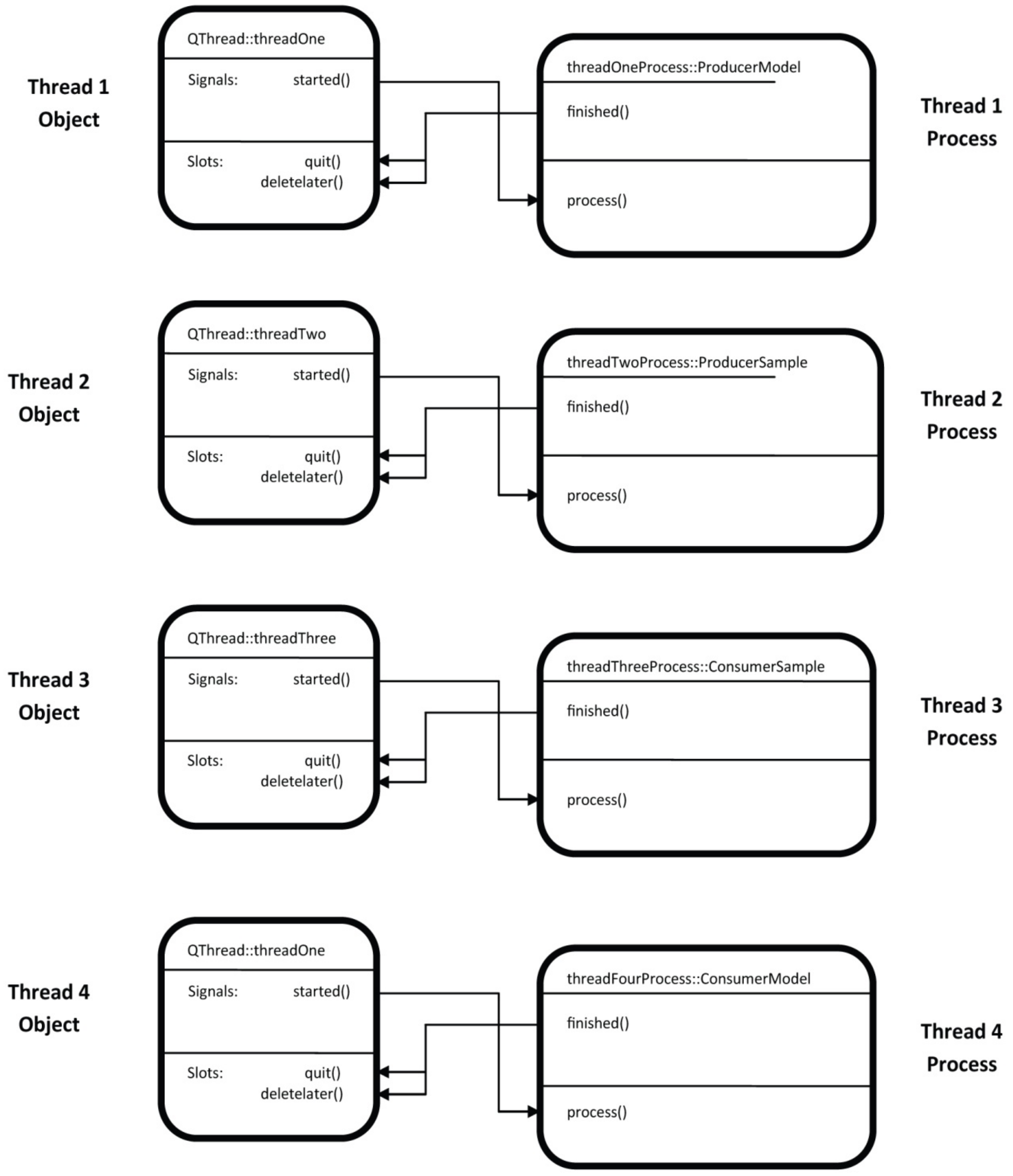


Figure 21: Signals & Slots, Connecting Threads and Processes

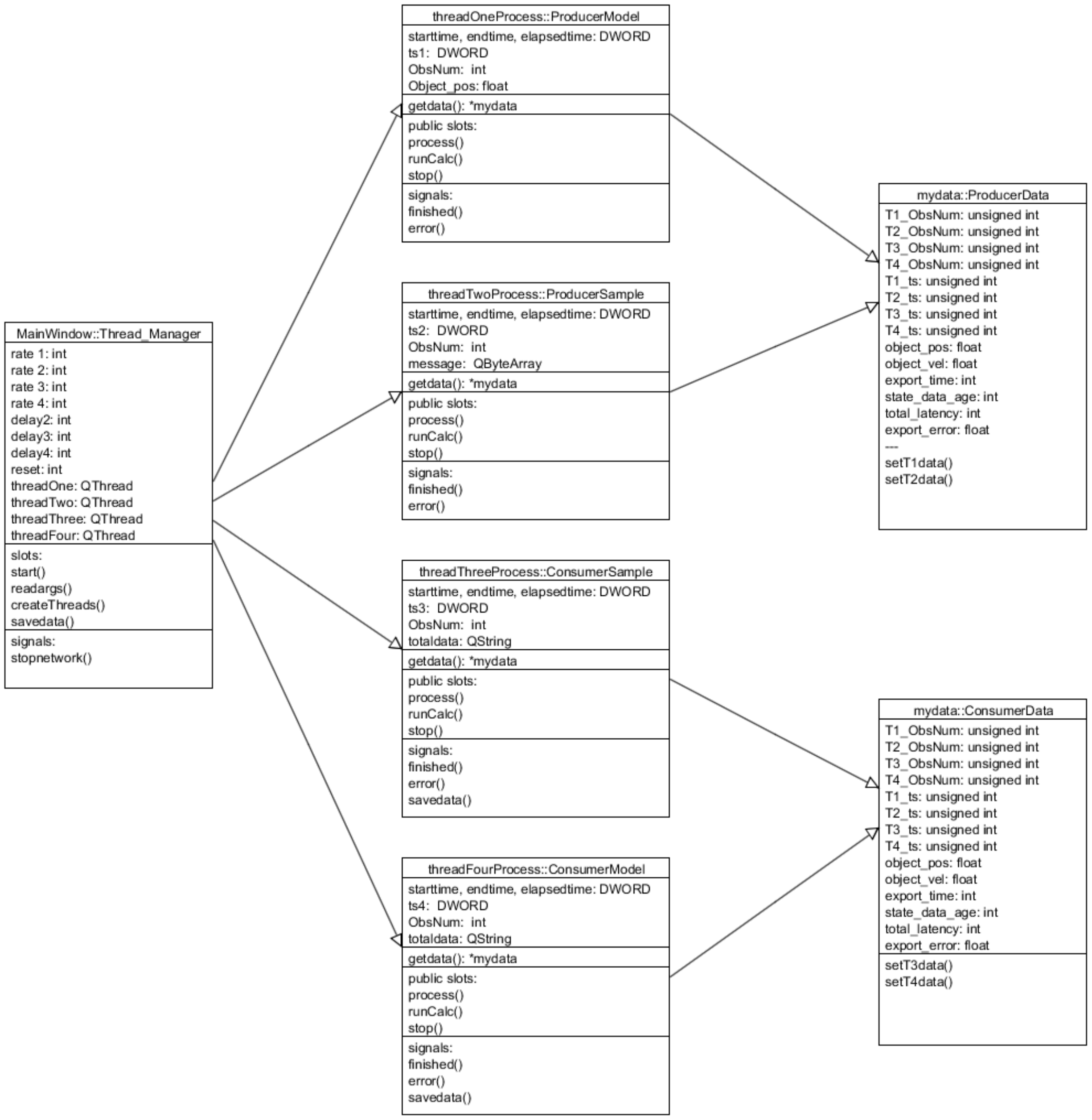


Figure 22: UML of Thread Manager and Thread Processes

Bibliography

Banks, J., Carson, J., B.Nelson, & Nicol, D. (2001). *Discrete-Event System Simulation.* Prentice Hall.

Belanger, J., Venne, P., & Paquin, J. (2010, March 3). *The What, Where and Why of Real-Time Simualtion*. Retrieved from Opal RT Technologies: http://www.opal-rt.com/sites/default/files/technical_papers/PES-GM-Tutorial_04%20-%20Real%20Time%20Simulation.pdf

Bottazzi, M., & Salati, C. (1991). Processes, Threads, Parallelism In Real-Time Systems. *CompEuro '91 Advanced Computer Technology, Reliable Systems and Applications 5th Annual European Computer Conference Proceedings.*, (pp. 103-107).

Cavitt, D. O., Maly, C. &., & K.J. (1997). A Performance Monitoring Application For Distributed Interactive Simulations (DIS). *Simulation Conference*, (pp. 421-428).

Churchill, E. F., Snowdon, D. N., & Munro, A. J. (2001). *Collaborative Virtual Environments: Digital Places and Spaces for Interaction.* Verlang: Springer.

Delaney, D., Ward, T., & McLoone, S. (2006). *On Consistency and Network Latency iin Distributed Interactive Applications: A Survey--Part I.* Massachusetts Institute of Technology.

Dewar, J. A., Bankes, S. C., Hodges, J. S., Lucas, T., Saunders-Newton, D. K., & Vye, P. (1996). *Credible Uses of the Distributed Interactive Simulation (DIS) System.* Santa Monica : Rand Corp .

Fritzson, P. (2003). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press.

Hodson, D. D. (2009). *Performance Analysis of Live-Virtual-Constructive And Distributed Virtual Simulations: Defining Requirements in Terms of Temporal Consistency.* Wright-Patterson Air Force Base: Air Force Institue of Technology.

Hofer, R. C., & Loper, M. L. (1995). DIS Today. *Proceedings of the IEEE , 83* (8), 1124-1137.

Murray, R. (2010). *DIS Overview and Version 7 Information*. Retrieved July 10, 2012, from Simulation Interoperability Standards Organization: http://www.sisostds.org/DigitalLibrary.aspx?EntryId=29288

Raja, P., Hernandez, J., Ruiz, L., Noubir, G., & Decotignie, J. (1993). A Software Architecture for Maintaining Temporal Consistency in a Distributed Real-Time Environment. 380-387. IEEE.

Ravindran, K., Sabbir, A., & Ravindran, B. (2008). Impact of Network Loss/Delay Characteristics on Consistency Control in Real-time Multi-player Games. *IEEE CCNC* , 1128-1133.

Roy E. Crosbie, P. (2007). High-Speed Real-Time Simulation. *First Asia International Conference on Modelling & Simulation* (p. 1). IEEE.

Singhal, S., & Zyda, M. (1999). *Networked Virtual Environments Design and Implementation.* New York: ACM Press.

Tanenbaum, A. S. (2007). *Distributed Systems Principles and Paradigms.* New Jersey: Pearson Education Inc.

Tolk, A. (2012). *Engineering Principles of Combat Modeling and Distributed Simulation.* New Jersey: John Wiley & Sons Inc.

Zalcman, L. (2004). *What Distributed Interactive Simulation (DIS) Protocol Data Units (PDU) Should My Australian Defence Force Simulator Have?* Edinburgh: Defence Science and Technology Organisation Edinburgh (Australia) Air Operations Div.

Zhou, S., Cai, W., Turner, S. J., & Zhao, H. (2003). A Consistency Model for Evaluating Distributed Virtual Environments. *The 2003 International Conference on Cyberworlds (CW'03)* (pp. 85-91). IEEE.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)* 21 Mar 2013	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* 03 Oct 2011 - 21 Mar 2013

4. TITLE AND SUBTITLE Estimating and Measuring Application Latency of Typical Distributed Interactive Simulation (DIS)-Based Simulation Architecture	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Drinkwater, Ryan L , Captain, USAF	**5d. PROJECT NUMBER** N/A
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-13-M-14
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of the Secretary of Defense Attn: Dr. Catherine Warner 1700 Defense Pentagon commercial #: (703) 697-7247 Washington D.C. 20301 e-mail: catherine.warner@osd.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** OSD
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
One of the challenges in a distributed virtual environment stems from the requirement to simultaneously execute the simulations in real-time to support human interaction, in conjunction with maintaining a consistent view of the shared simulated environment. Maintaining a consistent set of simulation state data in the presence of network latency is difficult if individual data items are updated frequently. The principle application of DIS-based simulation environments has been in the domain of training where a consistent view or its correctness is often judged in subjective terms such as the simulation looking and feeling correct. New application areas for these systems are emerging in the analysis and test domains. For these domains, quantifying shared state consistency in terms of overall distributed application architecture is desirable. This research effort will investigate and validate methods to calculate and measure the latency effects that consider the structure of the applications themselves. Additional latencies introduced due to the software architecture may significantly affect the consistency of the simulation. An improved understanding is beneficial to the Air Force where real-time distributed simulations used for the purpose of analyzing the systems they simulate and the support of live test events.

15. SUBJECT TERMS
Real-time Distributed Simulations, Software Architecture Latency, Simulation Consistency

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Douglas Hodson, Ph.D, AFIT/ENG
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*
U	U	U	UU	95	(937) 255-6565 (Douglas.Hodson@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18