RTI for Support for Real-Time Simulation

By

Dan Bleichman

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

M.A.Sc in Electrical Engineering

Department of System and Computer Engineering

Carleton University, Ottawa, Ontario

March 2006

Your file Votre référence
ISBN: 978-0-494-23328-3
Our file Notre référence
ISBN: 978-0-494-23328-3

Table of Contents

List of Figures

Chapter 1: Introduction

High-Level Architecture (HLA) is the current IEEE standard [1] for creating and managing distributed simulations. This standard is implemented by the Run-Time Infrastructure (RTI), which is a software package that can be executed using various hardware architectures and operating systems. When the RTI is used by individual simulators or modules of a simulator, a larger distributed simulation becomes possible. When human in the loop or hardware in the loop are involved there is usually a requirement for the simulation to execute in real-time. For a real-time distributed simulation to be effective, certain performance requirements must be met, requiring optimal use of the network as well as computing hardware. These performance requirements can be achieved by tuning, optimization or optimal utilization of the RTI. The correctness of real-time processing depends upon the time at which the result is produced, as well as the logical correctness of the computation. If the timing constraints of the system are not met, system failure is said to have occurred." Drake [2] has added: "It is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behavior requires that the system be predictable". "Hard" real-time are those discussed above, while "soft" real-time systems have less stringent requirements.

The following are the performance characteristics for an RTI simulation:

- Low Latency: Latency is the time between the onset of an event in one subsystem or module of a distributed simulation and awareness of that event in another. The elapsed time should be small enough for the simulation to function properly.
- Bounded latencies under reasonable processor utilization.

- Bounded latencies under reasonable network utilization.
- High throughput (in terms of messages per second) not an absolute requirement for real-time, but usually guarantees stable and predictable behavior below the throughput limits.
- The presence of a reliable message-receive mechanism (true as well for non-real-time RTIs) – messages are not "dropped" even when the federate is busy. This may not be required for certain applications.
- Time management that is part of the HLA standard (described in chapter 2) – beneficial for a real-time RTI as for a non-real-time RTI as long as it does not conflict with the previous requirements.

In this research, which was conducted at Carleton University during fall of 2004 and winter of 2005, three performance-improvement techniques were tested:

1) converting an RTI to a multithreaded process model
2) data-bundling
3) selection of a connectionless network protocol.

In addition, the performance cost of time management was measured, and improved performance measurement procedures were introduced and utilized through this research. The most significant findings from this research are the following:

- A multithreaded (asynchronous) RTI reduces latency over a single-threaded RTI, especially when additional computations have to be performed by the federate.
- A multithreaded (asynchronous) RTI provides more consistent message latencies than a single-threaded RTI, especially when additional computations have to be performed by the federate.

- Data-bundling significantly improves data throughput but increases message latency. However, the selection of appropriate bundling parameters is crucial.
- Under heavy RTI and network loads, reliable message delivery produces lower latency than unreliable message delivery. In less loaded conditions, unreliable message delivery generally yields lower latency.
- A multithreaded (asynchronous) RTI reduces general processor utilization over a single-threaded RTI, allowing execution of more federates on a single processor.
- The existing "logical" time-management algorithm restricts RTI performance to the point that real-time performance becomes impractical.

The most significant contributions of this research are the following:

- Demonstrated the performance advantages of multithreaded asynchronous RTI.
- Explained and demonstrated data-bundling as a way to increase message throughput.
- Demonstrated the different performance of reliable (TCP) and unreliable (UDP) data transfer in the presence of additional network traffic.
- Introduced an additional RTI benchmark that models more realistic conditions by sending and receiving concurrently.
- Introduced computing load and network load to standard HLA benchmarking.
- Proved that for real-time federations with 8 or more federates, time management may be impractical.

Chapter 2 provides conceptual background information about the HLA, the RTI, process models, data-bundling and HLA time management. Chapter 3 summarizes current RTI performance research. Chapter 4 first presents the five main questions this research will

address and then provides justification for why those questions should be answered. Chapter 5 first analyses in the processes involved in RTI message send and receive operations. Secondly, it presents a detailed description of a multithreaded RTI that has been developed in this research. Section 5 goes on to present the test procedures used in this research, and a theoretical analysis for potential performance enhancing techniques. Chapter 6 presents and analyzes the results for the following 9 experiments:

1. Single vs. Multithreaded RTI–One-Way Message Latency
2. Single vs. Multithreaded RTI–Average Receiver Throughput
3. Single vs. Multithreaded RTI–Average Sender Throughput.
4. Single vs. Multithreaded RTI–Average Loop-Back Throughput
5. Data-Bundling vs. No Data-Bundling–Average Sender Throughput
6. Data-Bundling vs. No Data-Bundling–Average Receiver Throughput
7. TCP vs. UDP–Average One-Way Latency
8. TCP vs. UDP–Average Receiver Throughput
9. Time-Advance Grant Throughput over LAN

Conclusions and ideas for future work are presented in chapter 7. Additional technical background information is provided in appendix B.

Chapter 2: Conceptual Background

2.1 High-Level Architecture (HLA)

"The High-Level Architecture is a standard framework that supports simulations composed of different simulation components" [3]. The HLA was originally developed by the Defense Modeling and Simulation Office (DMSO) of the U.S. Department of Defense (DoD) to meet the needs of defense-related modelling and simulation projects. Currently two separate HLA specifications exist: The earlier DMSO specification [4] and the later IEEE 1516 [1]. The HLA was developed in order to facilitate the *reusability* of simulation models in different simulation scenarios and applications. In addition, the HLA supports *interoperability*, which is the ability to combine component simulations (federates) on heterogeneous distributed computing platforms, often with the goal of real-time operation. The distributed federates are referred to as a federation. For the federation to function effectively, three components are required [3]:

1. HLA Rules: The rules ensure proper interaction of federates in a federation and describe the responsibilities of federates and federations.
2. Interface Specification: The interface specification defines Run-Time Infrastructure (RTI) services and interfaces, and identify the "call-back" functions that each federate must provide.
3. Object Model Template (OMT): The OMT prescribes the format and syntax for recording federation specific information and establishes the format of the following key models:
   i) Federation Object Model (FOM)

ii) Simulation Object Model (SOM)

iii) Management Object Model (MOM)

Two data types are defined by the HLA specification:

1. Attribute updates are used to transmit the state of objects owned by a federate (for example, a vehicle's attributes might include position, orientation and speed). Attribute updates are sent only when attribute values change.
2. Interactions are used to notify federates about events generated or noticed by other federates (for example, a torpedo fired by a submarine federate).

Each federate publishes the attributes and interactions it can send and that can be of interest to other federates. Each federate subscribes to attributes and interactions published by other federates and that are of interest for that federate.

2.2 Real-Time Infrastructure (RTI)

Real-Time Infrastructure is a collection of software that implements the RTI interface specification [4], [1]. RTIs are available from several commercial vendors, usually in the form of software libraries with which the federate developer can link his/her federate code. . Though all RTIs implement the HLA specification (either DMSO or IEEE 1516) partially or in full, they differ significantly in their internal architecture and other implementation details. The source code for an RTI is available from Georgia Institute of Technology [5] for research purposes, and as such, can be modified to address different requirements such as improved performance or added features. This source code is commonly referred as Federation Development Kit (FDK). The DMSO HLA specification was used in this research. Hence, all references to HLA services and

functions in this paper refer to their DMSO HLA definition. The services provided to the federates by the RTI software are:

1. Federation management: Creation, dynamic control, modification and deletion of federation execution.
2. Declaration management: Federates declare to the RTI their desire to publish and/or subscribe to object-state attributes and interactions.
3. Object management: Each object is assigned an ID that is used by the federate in order to transmit the object's state attributes.
4. Ownership management: Allows federates to transfer ownership of object attributes or complete objects since only the owner of a specific object instance may publish value for that instance.
5. Time management: Provides time-ordered exchange of events among federates in a federation.
6. Data Distribution Management (DDM): Supports efficient routing and distribution of data by abstract regions.

In this research only the following services were used: Federation management, declaration management, object management and time management. DDM and Ownership management are outside the scope of this research. Multicasting had not been used throughout this research since it is still not reliably implemented across WAN.

An HLA federation consists of between two and N distributed federates. Each federate is implemented as a user module that consists of software (code) and sometimes dedicated hardware as well. The federate interacts with other federates using RTI services

exclusively. As illustrated in Figure 1, each federate's code (user code) generates calls to it's Local RTI Component (LRC). The LRCs interact through an RTI-specific interface (which unfortunately implies that an RTI from vendor A cannot interoperate to an RTI from vendor B, and might not even interoperate with a different version of the same RTI). The LRC on the other federate/federates then invokes callbacks in the federate ambassador. The federate ambassador is essentially just a set of user (federate developer) implemented functions (callbacks) that are predefined by the HLA specification. The RTI can be configured and customized in many ways through a configuration file called the RID file (for instance, Data-Bundling configuration is set using this file).

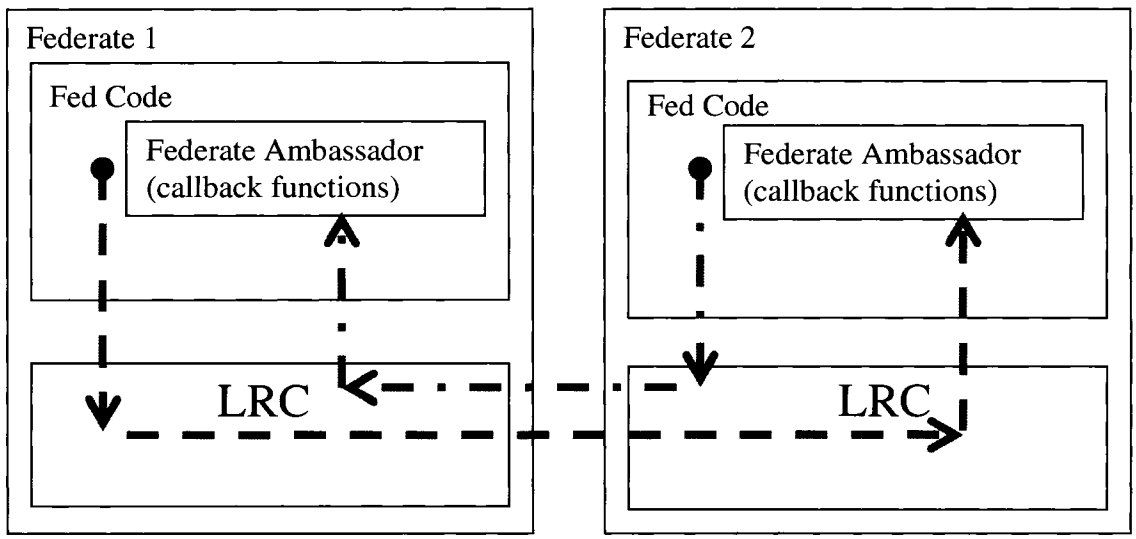


Figure 1–HLA Federate Implementation

2.3 RTI Process Models

The RTI's process model has a significant impact on its performance. The process models currently used in existing RTIs are introduced here. This introduction is based on chapter 2 of Karlsson [6].

The process model of an RTI determines how calls and callbacks are managed. As a consequence, the process model determines how the RTI and the federate share the processor. The HLA standard does not specify exactly how the process model of an RTI must work and vendors are currently using three approaches:

1) Single-threaded:

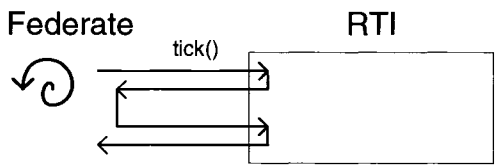


Figure 2–Single-Threaded RTI

In a single-threaded process model the federate must call *tick()* periodically to allow processing to be done by the RTI. The RTI uses the same thread of execution as the federate. The RTI processing may result in a call back to the federate. The federate processes the callback and then the RTI may do some more processing. Finally, control of the execution of the thread is returned to the federate. "In such RTIs, various problems can occur if a federate does not call *tick()* often enough. Incoming network packets may queue up, and when buffers fill, packets may be dropped. Of course calling *tick()* too often could mean starving the federate of CPU cycles" Granowetter [7]. This process model is the one that was used in the first version of the DMSO RTI [4], and prior to this research, is the only model supported by the FDK [5]. Whenever a tick call is issued by the federate, CPU time is granted to the RTI to process. Callbacks are delivered to the federate as part of the *tick()* call.

2) Asynchronous double-threaded:

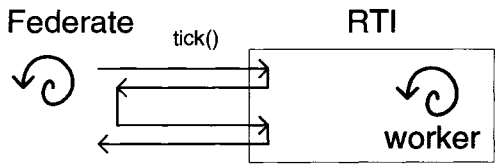


Figure 3–Asynchronous Double-Threaded RTI

In an asynchronous double-threaded process model the RTI uses one or more worker threads that run independently of the federate's thread(s). Thus, the RTI does not need to be granted CPU time since it uses its own internal thread(s) to perform internal processing. Despite the presence of RTI worker thread, callbacks are still delivered to the federate during the tick call. This process model is currently the most popular, used by RTIs such as DMSO NG-1.3 [4], MAK [8], and most versions of Pitch [9].

3) Asynchronous multithreaded:

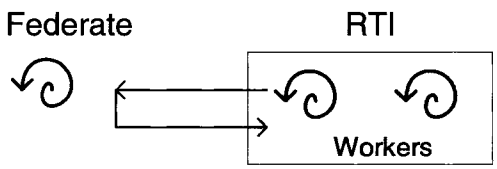


Figure 4 –Asynchronous Multithreaded RTI

In the multithreaded process model, the RTI uses one or more worker threads that run independently of the federate's thread(s). The thread performs the RTI's internal processing as well as delivering callbacks to the federate. The federate does not need to call tick since the worker threads deliver the callbacks. "This means that a callback can be delivered at any time and not only when the federate calls tick, thereby making the tick call obsolete" [6]. This process model was introduced by Pitch in their new pRTI [9]

product in late 2004. An asynchronous multithreaded RTI is not "thread-safe". At any time multiple callbacks can be invoked. Furthermore, several threads can invoke the same callback at the same time. It is up to the federation developer to employ safe multithreading techniques, especially prevention of concurrent access to variables and data structures using semaphores.

2.4 Data-Bundling

Data-bundling reduces the number of IP packets throughout the federation. Instead of allocating an individual IP packet for each attribute update or interaction and sending it immediately upon being constructed, data-bundling allows individual transactions to "pile up" in an RTI buffer for either a given amount of time or until the buffer reaches a predefined size. Then the buffered transactions are sent in a burst cycle while trying to utilize as few IP packets as possible. Data-bundling can improve efficiency as explained in section 5.6.

2.5 Message Delivery Modes

The HLA specifies two message delivery modes: "best-effort", in which reliable delivery is not guarantied and "reliable", in which the infrastructure (network layer and/or RTI) ensures message delivery. The RTI uses a federation file (commonly referred as .fed file) to specify the desired message delivery mode for each given attributes or interaction. There is assumed to be a trade-off between reliability and performance. I.E. If message delivery is reliable, lower message throughput is expected. RTI implementations often

use User Datagram Protocol (UDP) for "best-effort" and Transmission Control Protocol (TCP) for "reliable" message delivery.

2.6 Time Management in RTI

The HLA time management determines the advancement of each federate along the federation time axis. "The RTI provides an optional time management service to coordinate the exchange of events between federates. In some situations, it is appropriate to constrain the progress of one federate based on the progress of another. In fact, any federate may be designated a *regulating* federate. Regulating federates regulate the progress in time of federates that are designated as *constrained.* In general, a federate may be "regulating", "constrained", "regulating and constrained", or "neither regulating nor constrained". By default, federates are neither regulating nor constrained." [4] A federate that is time-regulating may associate some of its activities (such as updating object attribute values and sending interactions) with time-stamps on the federation wide time axis. A federate that is time-constrained is guaranteed to receive time-stamped notifications in time-stamped order. Time-management ensures coordination between time-regulating and time-constrained federates in a simulation execution.

The HLA time management guarantees that:

- A federate receives messages in a time-stamped order.
- A federate can't receive a message in its past.

This means that:

- At any instance during federation execution, different federates can be at a different logical time.

- The duration of a single logical time unit can vary in absolute time.
- The federation advances in time according to the **slowest performing** time-regulating federate.

Chapter 3: State of the Art in RTI Process Models and Performance Evaluation

The state of the art in RTI process models and performance evaluation is based on current non-restricted publication in this field.

3.1 Process Models

Karlesson [6] lists the following performance characteristics of the three process models:

- In a single-threaded RTI no effort is spent on context-switching, i.e. to "wake up" threads.
- The RTI might starve and stop responding to calls from other federates if federate code is busy with heavy processing.
- An asynchronous RTI never starves since all internal processing is done in a separate thread.
- Multithreaded RTIs have better responsiveness–that is, lower latency.
- Unlike single-threaded and asynchronous RTIs, multithreaded RTIs are not thread-safe. Federate developers must deal with concurrent access to data structures since a callback may occur at any time.
- The single-threaded process model requires extensive tuning and knowledge of the internal workings of the code of both the RTI and each federate and the environment where the federation is executed (hardware, software, OS, network topology).
- In some cases single-threaded RTIs can provide the best throughput due to minimal overheads (context switches).

- Generally a multithreaded RTI will provide the best throughput, though sometimes at the cost of CPU time.

The first and only commercial asynchronous - multithreaded RTI was introduced by Pitch AB in late 2004 and after most of the work for this study was complete. Though detailed information about its architecture is not yet available, it should be assumed that it functions in a similar way to the modified multithreaded RTI that was developed in this research. Limited benchmark information about the Pitch RTI is available but it was run on a different hardware platform than was used in the current research: a 2 GHz Intel Pentium 4, 512 MB RAM using Windows 2000 Server while this research used a much slower hardware (0.9 GHz Athlon with 192 MB RAM) and Linux OS. The Pitch RTI is also JAVA-based so direct performance comparison with the C/C++-based RTI used for this study is not possible since a JAVA implementation is executed using a JAVA virtual machine and hence is inherently slower. Pitch doesn't state how their benchmarking was done, how many senders and receivers where used during the attribute update throughput test, nor how their latency benchmark was performed. In addition, they provide no comparison to a similar single-threaded RTI.

Other currently available RTIs are either single-threaded or asynchronous dual threaded.

3.2 Measuring Latency

The *BmLatency* [10] benchmark was developed by DMSO as a standard benchmark to allow comparison between RTIs. A similar test procedure is used in this research and is described in section 5.3.1. It consists of a round-trip latency in which the sender samples the send time, sends a message to the second federate, which returns the

message to the sender, which records the arrival time. The round-trip time is the arrival time – send time. Message latency is half the round-trip time.

Fitzgibbons [11] and Knight [12] base their latency benchmarks on the standard DMSO *BmLatency* benchmark with minor modifications.

Knight et al [12] conducted variants of the DMSO *BmLatency* test. Their first variant is a round-trip latency using a limited frequency of 100 Hz. Their second variant used an unlimited frequency. I.E. As soon as the round trip was complete, the next round trip was initiated. Knight et al [12] state, "When two RTIs show equal latency, the one that can sustain a higher update rate is more desirable." Their results show that in most cases the relationship between latency and update rate is as expected (Update_rate = 1/Latency) but in some cases the results are inconsistent. They explain this by congestion occurring in the "returnee" federate network interface card (NIC).

Burks et al [13] describe a one-way latency test in which the send time of an attribute is embedded in one of the attribute fields. When the attribute update is received by the "receive" federate, it is compared to the local system clock and the latency is the time delta. Both the sender's federate machine and the receiver's federate machine are synchronized using the NTP server. Experience with NTP over LAN suggests time-synchronization accuracy of 1-2 milliseconds while one-way message latencies are typically smaller than 1 millisecond. They do not mention any additional compute or network loads.

Nemeth [14] benchmarked the message latency of several RTIs and performed an analysis based on the amount of packet overhead required by each RTI. Though his analysis shows that the RTI with the largest message latency also has the most overhead

bytes, he can't explain the large performance difference between RTIs; when the extra message overhead added only 10% to the size of the message, the measured latency for the RTI was much greater than 10% when compared with the other RTI.

Drake [2] suggests adding proxy federates in order to increase both message load and network load throughout the benchmark, however, authors have not implemented their suggestion.

3.3 Measuring Throughput

The DMSO *BmThroughput* [15] federate is designed to measure update throughput. With minor modifications it can be used to measure HLA interaction throughput as well. It measures throughput between two or more federates. Federates are designated as being either senders or receivers. The senders record the time needed to send a fixed number of updates, and the receivers record the time it takes to reflect those updates (or the time it takes to receive the interactions). These measurements can be repeated over a number of cycles. The resulting metrics are updates/sec and reflects/sec (or send_interactions/sec and receive_interactions/sec). In addition, the size of the update (or the size of the interaction) can be given as a parameter.

Fitzgibbons [11] used the standard DMSO *BmThroughput* test to benchmark RTI throughput for several RTIs including the FDK and DMSO NG1.3v4 (with and without data-bundling). They performed sender and receiver throughput tests with federations of 2, 4 and 8 federates. Unfortunately they do not specify how many federates acted as senders and how many as receivers for each run. Their results usually suggest better sender throughput than receiver throughput:

- The FDK had four times the sender throughput of DMSO NG1.3v4 RTI.
- The FDK had three times the receiver throughput of DMSO NG1.3v4 RTI.
- For two federates, the FDK demonstrated better sender throughput than receiver throughput.
- For a larger number of federates (4 or 8) sender throughput equaled receiver throughput.
- DMSO NG1.3v4 RTI provided a message throughput 3–4 times higher with data-bundling than without it. The data-bundling parameters weren't specified.

Knight, Corder & Liedel [16], Knight, Liedel & Klinner [17] and Knight, Liedel, Klinner, & Steele [12] used similar test procedures to Fitzgibbons [11]. They experimented with both attribute updates and interactions. In addition, they used multiple object types per federate, multiple attributes per object type, multiple instances of each object type, multiple benchmark interaction types and multiple parameters per interaction. They focused on a higher level of RTI implementation rather than the interaction between the RTI and network layer or interaction between the RTI and federate threads. Their tests demonstrate how to optimize object models for performance, regardless of RTI.

Knight [12] compared message throughput between "best-effort" delivery and "reliable" delivery over LAN using two federates (one sender and one receiver) using both MAK and DMSO NG1.3v4 RTIs. For both RTIs Knight [12] showed better throughput for reliable message delivery. They don't explain their results other than to say there was no message loss using "best-effort".

3.4 Cost of Time Management

Fitzgibbons [11] performed time advance grant benchmarking using the standard DMSO *BmTimeAdv* [18] federates as described in section 4.3.5. Their results show 2000 grants/sec per federate for a federation of 8 federates which request time-advance grant simultaneously (for a total of 16,000 grants/second) using a look-ahead value of one time unit. Their federation demonstrated a one-way latency of 0.3 milliseconds. Their result seems overly optimistic and does not agree with theory. Fujimoto's theory [19] suggests that at least NlogN messages (at 0.3 milliseconds each) have to be exchanged in order to fulfill a single time advance grant request. This implies that with N = 8 and one way message latency of 0.0003 seconds only 1111 grants/second are possible. Fitzgibbons [11] did not assess the cost of time management over a WAN using typical network delays.

Hung [20] using the DMSO RTI measured the performance cost of executing a federation in which time management is enabled on each federate but actual time management services were not used. During the tests all federates exchanged attribute updates. It was discovered that the federation performed considerably slower when time management was enabled than without.

Chapter 4: Research questions – Problem statement

This research examines performance enhancement techniques for RTIs in order to make RTIs suitable for real-time simulation. At an early stage of this research it became evident that existing performance measurement techniques as described in publications may not provide correct results under all conditions. In addition, an in depth examination of the RTI's time management mechanism raised questions regarding it's performance impact on the simulation. This is summarized by the following research questions:

4.1 Main questions:

1. By how much can the performance of an RTI be improved using a multithreaded architecture?
2. By how much can the performance of an RTI be improved using data-bundling? If yes, how should it be configured?
3. Can the performance of an RTI be improved by utilizing connectionless (UDP) or connection-oriented (TCP) network protocol?
4. Are the existing performance measurement techniques sufficient? If not, what improvements are required?
5. Can time management in a real-time federation be used while still achieving real-time performance?

4.2 Justifications:

1. Little research has been done on the relationship between the process model and the performance of RTIs; i.e. nothing that presents performance comparisons between RTIs that differ only in their process model.

2. Only one paper Fitzgibbons [11] presents RTI-throughput results with data-bundling but the authors do not state the configuration values (data bundling period and/or size of data bundled before data is sent). No theoretical analysis is included.
3. Current studies of performance of RTIs do not include network load.
4. Current measurements of message latency do not include computing load.
5. Current measurements of message throughput do not measure receiver throughput (though they claim to do so both in the DMSO *BmThroughput* benchmark [15] and in the US army SMDC/COLSA Corporation studies [12], [16], [17]) because the number of senders is always equal to (or fewer than) the number of receivers; the receivers are underutilized.
6. Current benchmarks do not simultaneously send and receive.
7. The performance cost of time management is not quantified.

4.3 These questions need to be answered because:

1. Multithreaded RTI can potentially reduce attribute update or interaction latency.
2. Multithreaded RTI can potentially improve concurrent send and receive operations.
3. It is important to look at message latency with "real-world" computing load.
4. It is important to look at message latency with "real-world" network load.
5. There may be performance effects of data-bundling configuration, and trade-offs between latency and throughput.
6. There may be performance costs associated with existing time management algorithms.

7. There may be performance effects to the selection "best effort" (UDP) message delivery over "reliable" (TCP) message delivery under "real-world" conditions.

4.4 Scope

For the purpose of this research a multithreaded RTI will be developed based on the single-threaded RTI from Georgia Institute of Technology [5] (FDK). The performance (message latency and throughput) of that multithreaded RTI will be measured against the performance of the original single-threaded version. Secondly the message throughput of the DMSO RTI NG1.3 [4] with data bundling will be compared to the message throughput of the same RTI without the data bundling. "Best effort" (UDP) message delivery performance (message latency and throughput) will be compared to "reliable" (TCP) message delivery performance using a UDP variant of the FDK [5]. The UDP variant will be developed for the purpose of this research based on the existing TCP only version. The performance measurements will be performed using improved performance measurement procedures that will be developed in this research as well. Finally the performance cost of time management will be measured for federations of several sizes over LAN using the DMSO RTI NG1.3 [4] and the standard DMSO *BmTimeAdv* [18] benchmark.

4.5 Research contributions

1. Performance aspects of multithreaded RTI are compared to a single-threaded RTI that is identical otherwise.
2. Effects of data bundling on message throughput are investigated.

3. Performance of "Best effort" (UDP) message delivery is compared with "reliable" (TCP) message delivery.
4. New RTI performance measurement procedures that introduce network load, processor load and concurrent message send and receive are proposed.
5. The performance cost of time management over LAN is quantified and explained.

Chapter 5: Test Design, Theoretical Analysis and Predictions

This chapter will first provide a technical analysis of the Operating System (OS) and network layer used in this research. Secondly the multithreaded RTI architecture will be described in detail. Next this chapter will provide a detailed presentation of the modified performance measurement procedures used in this research. The last part of this chapter presents a detailed step-by-step theoretical analyses of all the RTI's functions that will be benchmarked in this research, including predictions of expected results.

5.1 Relationship between RTI, OS and Network Layer

Except for the first three performance values below (for context switch latency, semaphore shuffle time and inter-process message passing), all analysis in this section is original.

5.1.1 Operating System

Red Hat Linux version 6.2 with kernel 2.2.14 was used. Typical performance figures were published by Heurch [21] for this kernel on similar hardware to that used in these experiments (x86 processor at 800 Mhz). These figures were:

- OS tick time: 10 milliseconds (system clock 100Hz).
- Context switch latency: 4 microseconds for a single process in the ready queue, 60 microseconds with 10 processes in the ready queue and linear for anything in between.
- Semaphore shuffle time: 5 microseconds.

- Inter-process message passing: 3 microseconds.
- Interrupt latency time: there are differing definitions for this term. Some define it as the time between when an external peripheral device generates an interrupt and the ISR (Interrupt Service Routine) executes the first command. In this paper a much more conservative definition is used which is: the time elapsed between when the hardware removes the currently running task from the CPU due to an interrupt and when that process or some other process gets the CPU back after the system returns from the ISR. Though a minimal ISR wouldn't introduce an interrupt latency of more than 13 microseconds (depending on the amount of work it has to perform and on external conditions such as hard interrupt load due to busy hardware), latencies of up to 2 milliseconds due to interrupt from a network device (for example Ethernet controller) are common with this kernel version. This phenomenon was significantly improved with the introduction of kernel 2.4.20 ("NAPI fix").

5.1.2 Network Layer

Two aspects of the IP network layer are relevant: message send and message receive. The following introduction to RTI message send and receive uses the information by Herrin [22] and Shanker [23] as a baseline.

Message Send

An outgoing message begins with an application system call to write data to a socket (in our case from the RTI). The data packet starts its way down the IP stack layers. At each layer, header information is added to the data buffer. Eventually the link layer tries to place the buffer in the send queue. This is not yet the hardware buffer. If there is space in the send queue, the link layer places the buffer there and invokes the device's transmit

function that will try to copy the buffer into the hardware's memory. If space is available, the packet goes into the network device hardware memory and is sent by the device. (As soon as the NIC finishes sending an Ethernet packet, it generates a hard interrupt.) If there isn't space in the network device the transmit function signals the OS (through a soft interrupt "trap") to invoke a kernel thread called *ksoftirqd* to send it later and the transmit function will return from the socket call. If there isn't enough space in the send queue, the RTI thread blocks. The *ksoftirqd* thread has a lower priority than the RTI thread so it will be scheduled only when the RTI blocks due to insufficient space in the send queue.

At this stage the *ksoftirqd* transfers all the data buffers from the send queue to the network device's hardware memory before returning control to the RTI thread. If the RTI thread blocked due to insufficient space in the send queue it will now be scheduled to complete the send and will return. If the *ksoftirqd* thread is busy with other work, such as handling incoming data, the RTI thread may wait longer before it resumes. In addition, when using TCP, the flow-control mechanism may block the RTI thread during the send procedure until the far side (network layer on the other federate's machine) indicates that it is ready to receive more data. In summary, calls to socket *send()* by the RTI may involve soft interrupts as well as process switches to the *ksoftirqd* thread. A hard interrupt is generated for each outgoing message.

Message Receive

The first packet to arrive in the network hardware memory generates a hard interrupt as soon as the packet finishes being "DMAed" (DMA – Direct Memory Access) into the processor's memory. Additional incoming packets can be "DMAed" since DMA

is a pure hardware operation. The hard interrupt first disables further interrupts from the network hardware and then generates a soft irq (irq – interrupt request). The soft irq processes up to a predefined number of incoming queued packets (for example, 300) and then enables interrupts from the network hardware. The interrupted process (probably the RTI application) is then rescheduled. If it was blocked on *receive()* during a socket call, it is resumed.

If there is a flood of incoming packets, the soft irq signals the kernel to schedule the *ksoftirqd* thread as soon as is practical (as soon as *ksoftirqd* can have the CPU). If this happens, as soon as the *ksoftirqd* thread is executed, it processes the remaining packets in the receive queue, interrupts are enabled and new received packets are placed into the available space.

If, even with the help of the *ksoftirqd* thread, the system is unable to keep up with the incoming data, additional packets are dropped. This is not likely with TCP since the flow-control "sliding window" mechanism is responsible for informing the sending side about the available space in the receive queue. The "Ack" (Acknowledge) packets that drive the "sliding window" mechanism are either sent by the soft irq or by the *ksoftirqd* thread, whichever processes the received packet up the network layers. With UDP on the other hand, many packets may be dropped.

In summary, when the rate of incoming packets is low, each packet will likely generate a hard interrupt. There is no risk of data dropping and the *ksoftirqd* thread is not required for the receive operation. Under moderate load, each set of incoming packets will generate only a single hardware interrupt. This is more efficient. The *ksoftirqd* thread may be scheduled but only when higher-priority threads such as the RTI are blocked.

Under heavy load due to a disabled receive interrupt, packets might be dropped using UDP.

From the network's point of view, as Shankar [23] suggests, a send operation generates greater processor utilization than a receive operation. One hard interrupt is generated and processed for each outgoing Ethernet packet (which may contain a single HLA message or a few) while only a single hard interrupt is generated for each sequence of incoming Ethernet packets. A well-tuned federate should be able to receive more messages in a given period of time than it can send.

5.1.3 RTI Processes

Different OSs handle user processes (or threads) in different ways. In this case (using Linux kernel 2.2.14) whether the RTI process or processes were scheduled as SCHED_FIFO or SCHED_OTHER, they should run for the full 10 milliseconds (the OS "tick" period) unless they are blocked by a call to *send(), receive()* or a voluntary *sleep()*. As was observed here and by other Linux kernel 2.2.X users Heurch [21], the *sched_yield()* function was ineffective. Furthermore, a call to *sleep()* with a sleep time of less than the OS tick time was not effective since the sleep time is rounded up to 10 milliseconds. This implies that if there are few RTI threads (federates) executing concurrently on the same CPU, they can not pre-empt one another before their time slice expires unless a running thread becoming blocked by performing a system call. Even when the process has nothing effective to do except polling the receive-socket queue by calling *tick()* it wouldn't release the CPU. This in effect wastes CPU cycles that could be used by other federates on the same machine. As would be shown later on, this

phenomena reduces performance for some RTIs but not for those that do not have to poll *tick()*.

5.2 Multithreaded (Asynchronous) RTI

The multithreaded RTI used in this research was developed from the FDK version 4.02 (The single-threaded RTI from Georgia Tech was described in chapter 2). The architecture of the multithreaded RTI is shown in Figure 5.

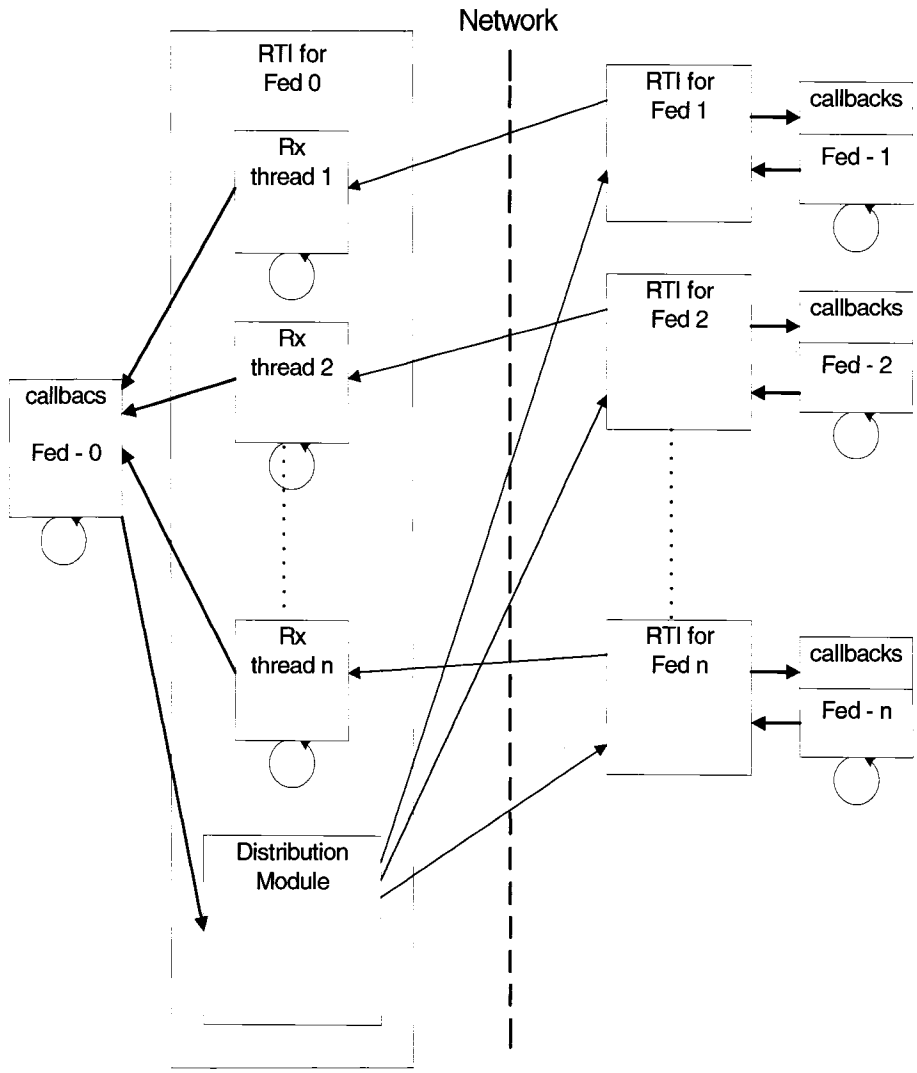


Figure 5-Multithreaded RTI

The multithreaded architecture was implemented using kernel-level threads due to their better (faster) responsiveness to external events (such as incoming data) comparing to user-level threads. There is a single user thread for each federate, which executes all the federate's internal code including RTI ambassador calls. This means that when a federate sends to other federates using the distribution module that simply places the message on the outgoing socket/sockets, the entire process, including placement of the packet on the network device hardware or at least on the link layer send queue (if the device's buffers

are full) – is performed within the thread. The user thread(s) is(are) scheduled using the "round-robin" policy; each one for a duration of a single time slice (10 milliseconds). This thread(s) may block due to 3 reasons:

1. The link layer's send queue is full.
2. The IP flow-control mechanism may block the thread during the socket-send process.
3. User code performs a system call that results in blocking.

In addition to the main federate's thread, there is a receive or "Rx" thread for each peer federate. This thread handles received messages from a single peer federate. The Rx thread has a real-time priority that is higher than the priority of the federate's thread. The Rx thread is blocked most of the time on a receive socket since there is one receive socket per peer federate. As soon as an incoming IP packet is delivered to that socket, the OS scheduler unblocks the thread. Then the Rx thread copies the message from the socket's buffer and invokes a callback function (the callback function is part of the federate ambassador). The Rx thread is allowed to invoke RTI ambassador calls from within the callback since there is no risk of an infinite loop. After the return of the Rx thread from the callback, if there are more messages pending on the socket, they are delivered in the same manner. If there are no more messages, the Rx thread will block again on the receive socket and the main federate thread is rescheduled. If there are many incoming messages on several receive sockets (such as when performing the receiver throughput benchmark with multiple senders) the Rx threads are scheduled according to round-robin policy; each one for a duration of a single time slice (10 milliseconds). This architecture is asynchronous since the main federate thread may be interrupted at any time by an Rx thread.

5.3 Performance measurement Procedures

A detailed description of the performance measurement procedures that have been used in this research is given below. The procedure to benchmark message latency was modified from the original DMSO's *BmLatency* procedure to include additional compute and network loads. The concurrent message send and receive is new. The senders and receivers throughput measurement procedures are closely based on the standard DMSO's *BmThroughput* procedure, The procedure used to assess the performance cost of time management is unchanged from DMSO's *BmTimeAdv* and is shown here last.

5.3.1 Latency Measurement Procedure

The latency measurement procedure is shown in *Figure 7.* Since adequate synchronization between the senders and receivers real-time clocks is not possible without GPS receiver hardware (NTP is subjected to network delays especially when the network is busy), One-way Latency was measured as the round-trip time divided by two. Though it is possible that message latency in one direction is always shorter (or longer) than in the other direction, since both sides using the same software and identical hardware it can be assumed that one-way latency is half of the round trip time. The timer was started immediately before the sender calls *UpdateAttributeValues()*. The receiver sends an identical update (a reply to sender) back when it "reflects" the update. The sender stops the timer immediately after it reflects the returning update.

In addition, a computing load was inserted between *tick()* calls at both ends to reflect the effects of different process models on RTI performance. An additional network load was injected using a separate node in order to better emulate "real-world" conditions. As in the standard *BmLatency* test, there is also an option to execute the tests with various

attribute (message) sizes. This latency measurement methodology is sufficient on operating systems where access to hardware clocks through simple system calls is possible (such as Linux or vxWorks – this is described in detail in appendix B1). When using an operating system such as MS Windows, which doesn't allow for direct access to hardware clocks and where the results returned by the system calls can be skewed by up to 10 milliseconds, a different strategy should be used. The problem can be overcome by altering the code on the sender federate as shown in Figure 6 where latency is measured over a large number of cycles.

```
record startTime
While(has to perform more cycles)
{
  send new attr update
  tick()
  While(didn't get reply)
  {
    tick()
    execute computing load
  }
}
record endTime
One Way Latency = (endTime - startTime) / (2 * number of cycles).
```

Figure 6 - sender federate code

As long as the number of performed cycles is large enough, the inaccuracy in system clock sampling will be insignificant.

One Way Latency = (totalTime) / (2 * number of cycles).

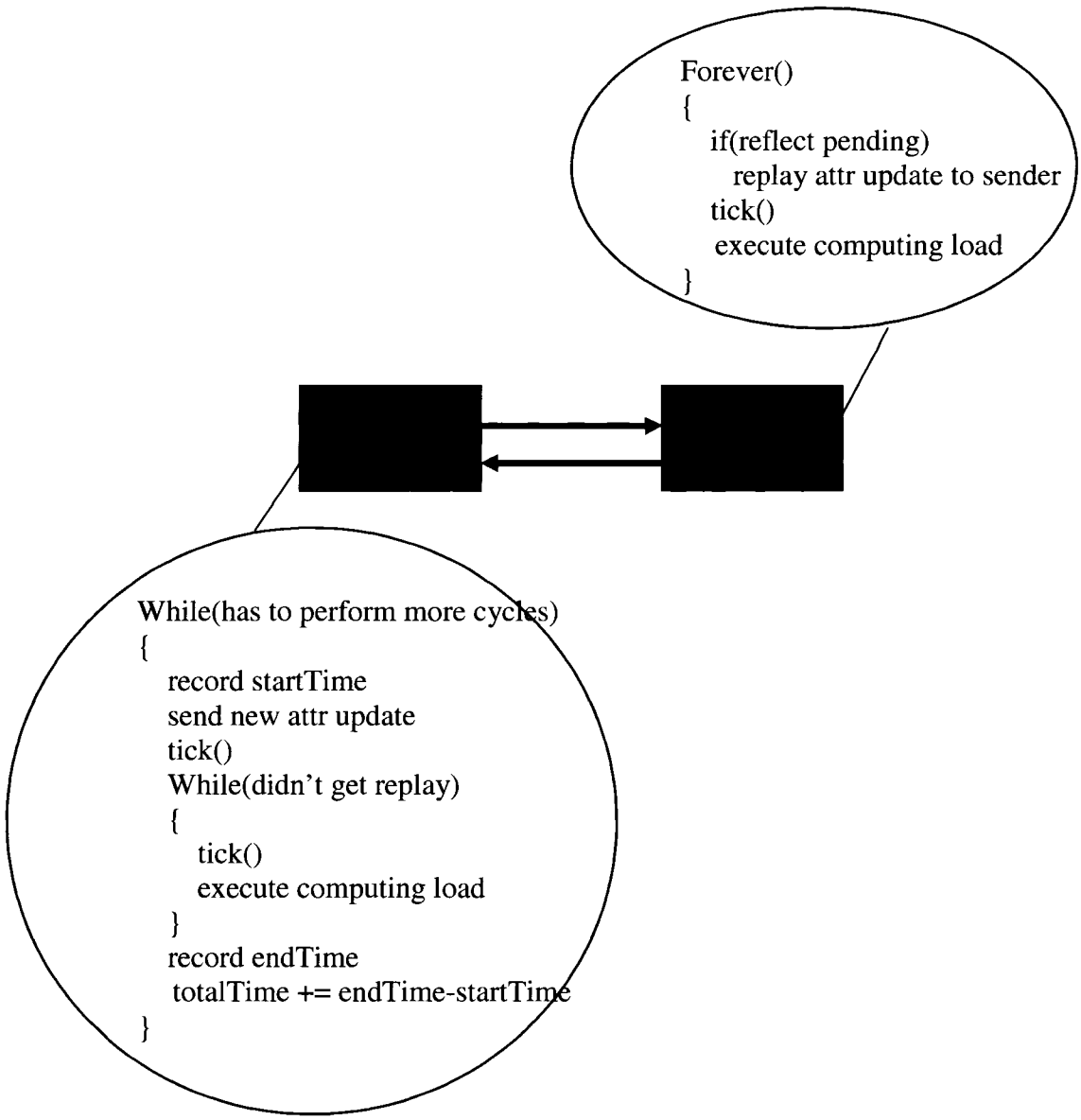


Figure 7–One-Way Message Latency

5.3.2 Sender's Throughput Measurement Procedure

The sender throughput test is essentially the standard DMSO *BmThroughput* benchmark.

As shown in Figure 8, the sender's throughput experiment consists of a single sender and a variable number of receiver federates. The sender records the time needed to send a fixed number of updates or interactions. An additional network load can be injected using a separate node in order to better emulate real-world conditions. As in the standard *BmThroughput* test, the test can use various attribute (message) sizes.

Throughput = number of sent updates / (endTime - startTime)

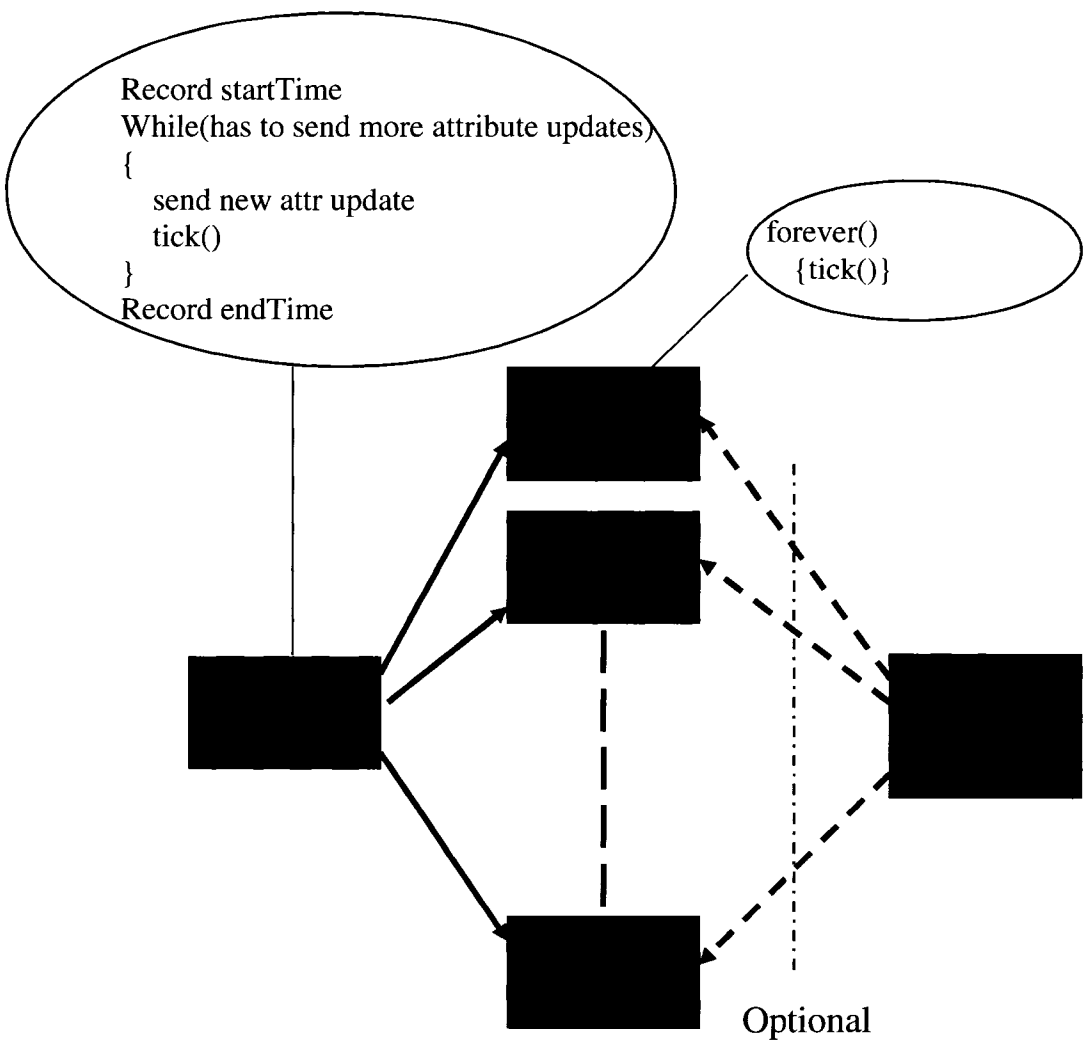


Figure 8–Sender's throughput

5.3.3 Receiver Throughput Measurement Procedure

In practice, a receiving federate must handle messages sent by several federates concurrently. The receiver throughput is measured using the setup shown in Figure 9. The receiving federate records the start time upon reception of the first attribute update or interaction and the end time when it receives K messages (K is given as a user parameter). Each sender sends K/N messages where N is the number of senders. An additional network load can be injected using a separate node in order to better emulate real-world conditions. It is expected that different process models would affect the receiver throughput.

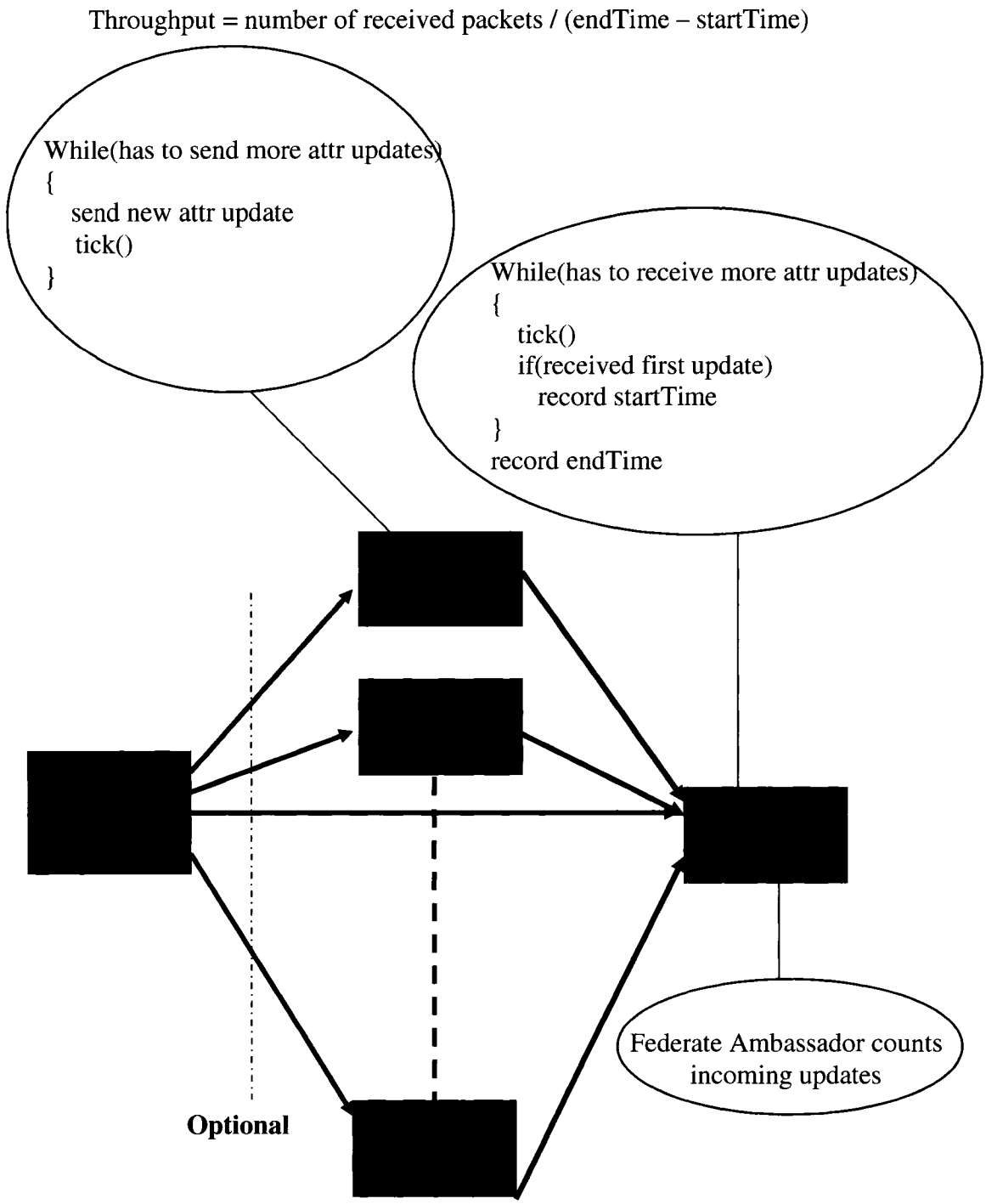


Figure 9–Receiver Throughput

5.3.4 Concurrent Send and Receive Throughput

Federates are frequently required to send and receive messages concurrently. In the test shown in Figure 10, the sender federate starts a timer and then distributes a number of

attribute updates or interactions to all "repeater" federates over time. The "repeater" federates replay the message to the sender upon receipt. The sender stops the timer after receiving a replay for all sent transactions. Concurrency occurs since the sender receives replays while sending new transactions. This experiment demonstrates a typical scenario where a single federate distributes an update regarding it's status/position/etc to all subscribed federates and all/most subscribers respond to it immediately and sending data back to that federate.

The sender initiates K (K is a user defined parameter) transactions. Each one is distributed to N "repeaters" and receives back K*N transactions.

Throughput = number of sent/received updates / (endTime - startTime)

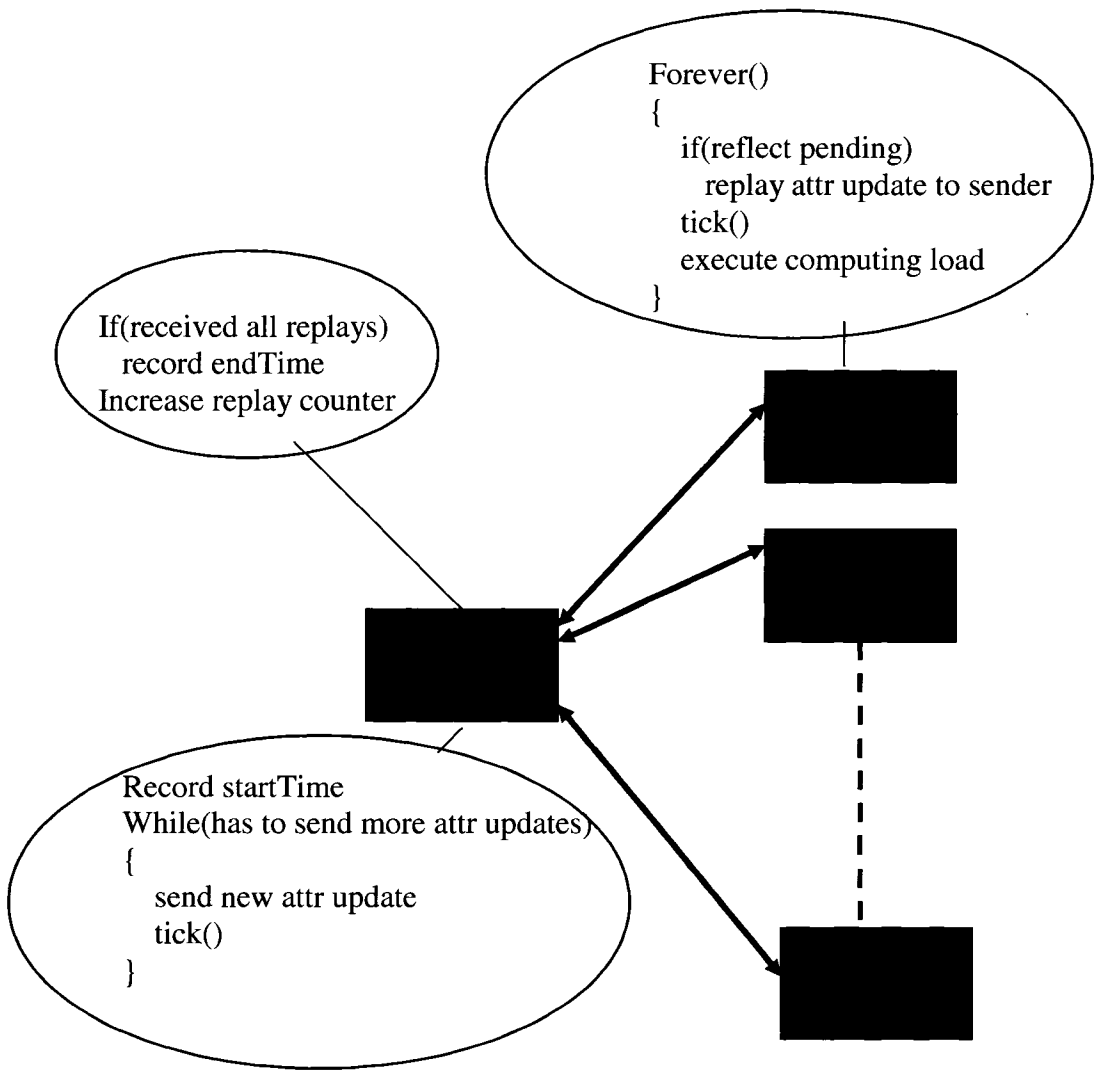


Figure 10–Concurrent Send and Receive Throughput

5.3.5 Time-Advance Grant Performance

The standard *BmTimeAdv* test was used to measure the time needed to advance a specified number of logical time units. The timer is started before generating the first "Time Advance" request and is stopped when the last "Time Advance" request is granted. All participating federates issue requests concurrently. This test requires both

low latency and good throughput to record high results (measured in grants per second). The test architecture is shown in Figure 11.

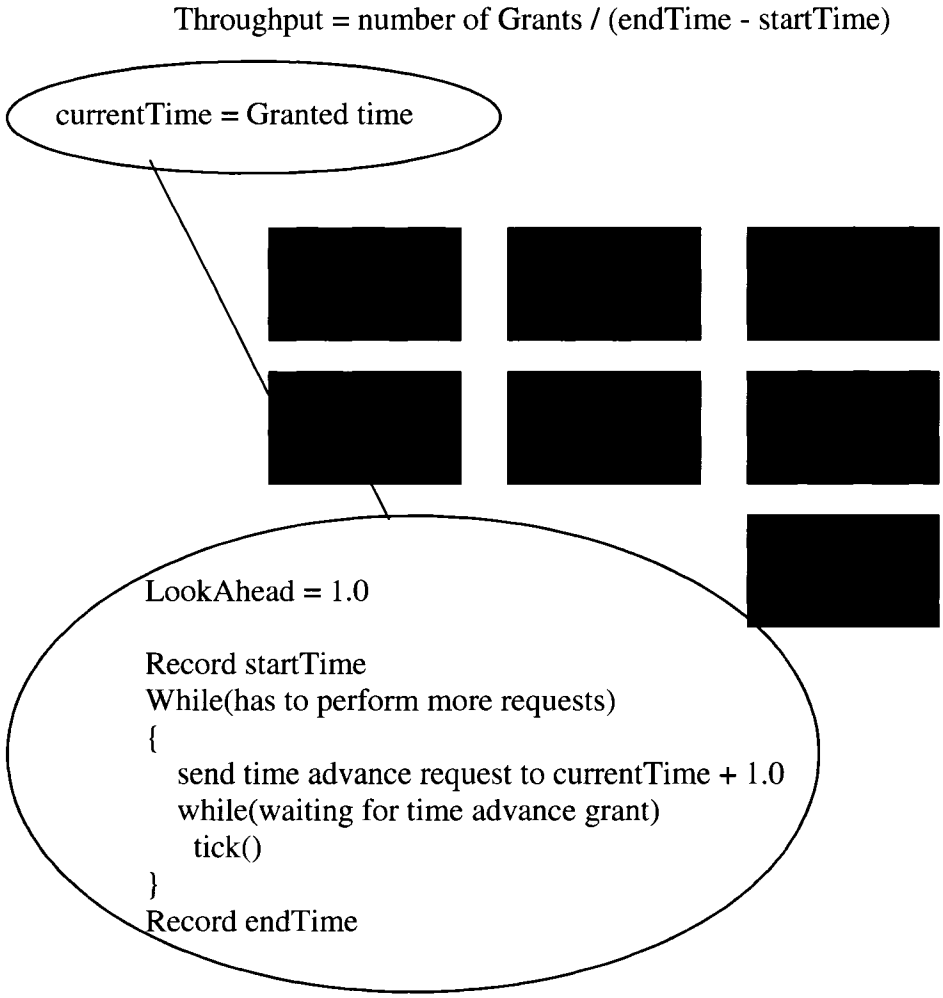


Figure 11–Time-Advance Grant Performance

5.4 One-Way Latency With and Without Computing Load for Single and Multithreaded RTIs

Section 5.4 provides a detailed analysis of one-way message latency. The first part (5.4.1) is a high level analysis. The second part (5.4.2 – 5.4.5) examines in detail the behavior of the various components that are involved in the process of sending a HLA

message and then receiving it back. The detailed analysis will point out the differences in behavior between single and multithreaded RTIs when used with and without additional computing load.

5.4.1 Round-Trip Delays from High-Level Perspective (Developers View)

Figure 11 shows the delays imposed by different components of a two federate federation when a single message is sent by federate F1 and then looped back to F1 by federate F2. This is a high-level description that does not reach deep into the network protocol nor does it deal with thread scheduling or software/hardware relationships (i.e. hard and soft interrupts). This description is based on an analysis (and Figure) provided by Knight [12] with some modifications and added details. It is applicable for both synchronous single-threaded RTIs and asynchronous multithreaded RTIs. In Figure 11, the local RTI component is denoted by LRC and the federate ambassador is denoted by FA.

The d's in Figure 11 denote different kinds of delays, as detailed below.

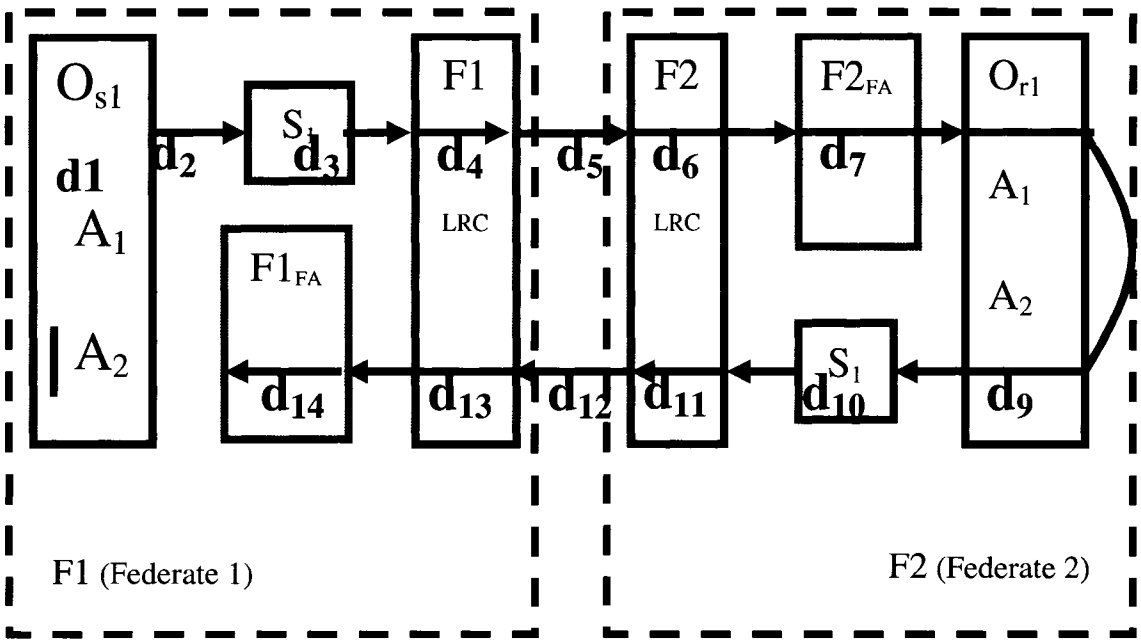


Figure 12–One-Way Latency from high-level prospective Knight [12]

d1 = Time F1 federate code spends accessing local object Os1 and getting the round trip start time Ts (federate code).

d2 = Time F1 federate code spends creating one Attribute Handle Value Pair Set (AHVPS) (S1) by calling (*createAttributeHandleValuePairSet()*).

d3 = Time RTI spends copying S1 from F1 federate code to F1 LRC (*updateAttributeValues()*).

d4 = Time F1 LRC spends processing and initiating delivery of S1. May depend on *tick()* (in synchronous RTI) and includes RTI bundling period and RTI data-bundling overhead.

d5 = Time network layer spends moving S1 from F1 LRC to F2 LRC. This includes the transmission medium and protocol overhead. May also include packet fragmentation and reassembly, retransmission delays, etc.

d6 = Time F2 LRC spends processing and providing S1 to F2 FA. May also include RTI bundle-splitting.

d7 = Time F2 spends processing S1 reflection in F2FA (user code) including copying reflected values to corresponding local returnee proxy attributes and queuing a return reflection to occur in F2 federate code.

d8 = Time F2 federate code spends processing return reflection queue and accessing local proxy object Or1.

d9 = Time F2 federate code spends creating AHVPS S1 with A2 return update stored in local proxy attribute buffer (*createAttributeHandleValuePairSet()*).

d10 = Time RTI spends copying S1 from F2 federate code to F2 LRC (*updateAttributeValues()*).

d11 = Time F2 LRC spends processing and initiating delivery of S1. May depend on *tick()* (in synchronous RTI) and includes data-bundling overhead.

d12 = Time network layer spends moving S1 from F2 LRC to F1 LRC. This includes network hardware and protocol overhead. May also include packet fragmentation and reassembly, retransmission delays, etc.

d13 = Time F1 LRC spends processing and providing S1 to F1 FA. May also include RTI bundle-splitting.

d14 = Time F1 (user code) spends processing S1 return reflection in F1 FA including getting the current round-trip time Te and calculating and storing the round-trip period (Te – Ts).

Asynchronous RTIs combine delays (d3, d4) and (d10, d11).

A low-level RTI-specific and network layer-specific analysis of the same round trip is shown in figures 13 (for a single-threaded RTI) and 14 (for a multithreaded RTI) and is described in sections 5.4.2 and 5.4.3 respectively. The red arrows represent flow of control and data while the black arrows represent flow of control only. Path 2 represents cases when actual data to send or to receive is available while path 1 will be executed when there is no data to be exchanged.

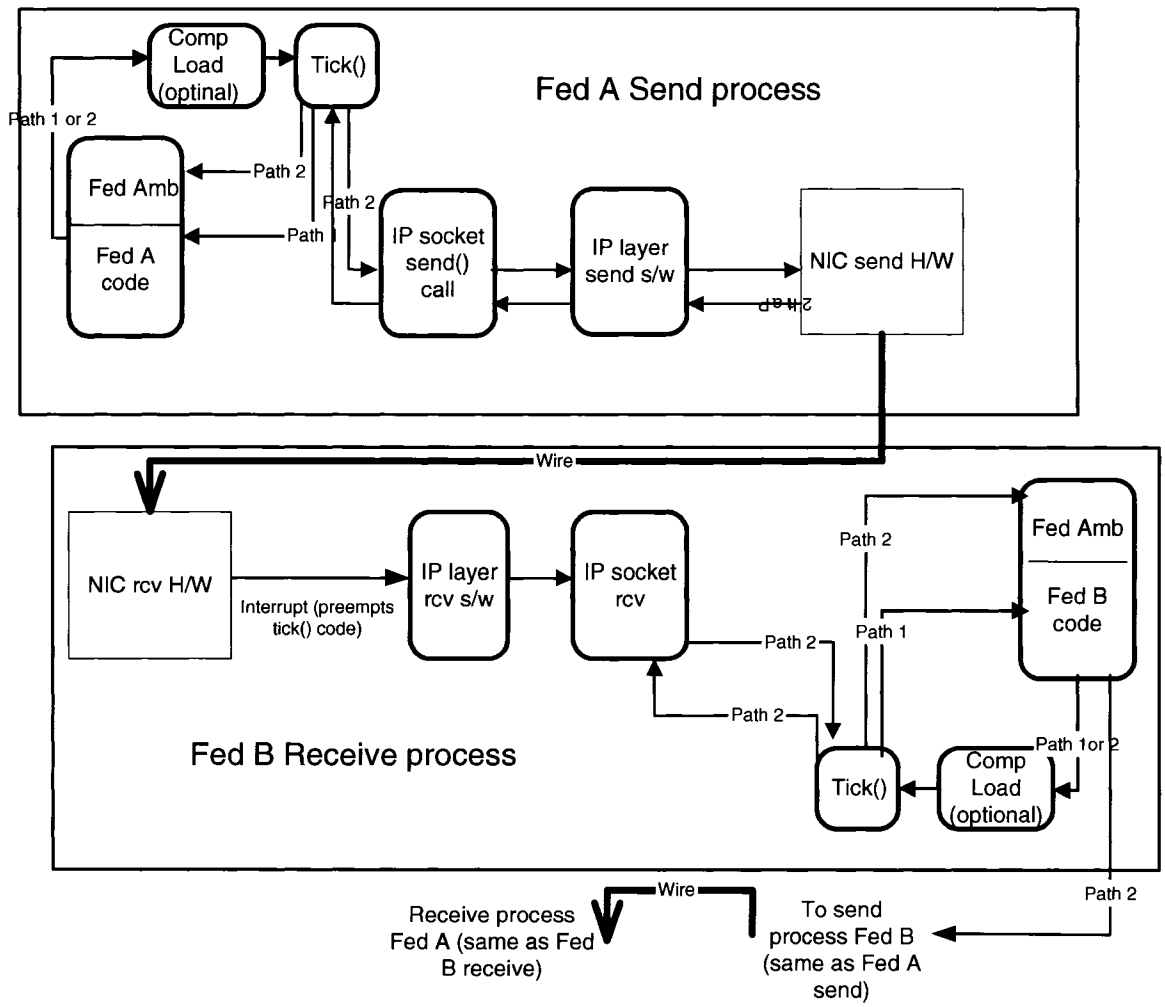


Figure 13–One-Way Latency Single-Threaded: Low-Level perspective

5.4.2 Single-Threaded RTI, No Computing Load TCP and UDP

The round trip of a single message from Fed A to Fed B and then back to Fed A is detailed below:

The attribute update message is created by Fed A code and the following sequence of events begins:

a. Send time is sampled (**T**send).

b. A call to send *updateAttributeValues()* is generated and a message (packet of data) is queued in the RTI outgoing message queue.

c. A call to *tick()* is issued and the *tick()* function dequeues the message and passes it down to the socket send function.

d. The packet travels in the context of the federate thread all the way down the IP layers. Since the network device is not busy the packet is copied to the network hardware and sent to the wire. The function returns. On the way up it checks for incoming messages (path 2). Since none is there it returns all the way up to the federate code.

e. Sender calls *tick()* again and again until there is an incoming message from Fed B.

f. Meanwhile, the attribute update travels through the wire, reaches the furthest network hardware, which DMAs it and generates an interrupt (DMA done interrupt).

g. The DMA done interrupt pre-empts the loop-back federate, which is also in a *tick()* polling loop. The hard interrupt generates a soft interrupt that delivers the message to the socket to enable interrupts.

h. The federate thread is scheduled again. During its current *tick()* call, while performing a *select()* on the receive socket, it finds the incoming message and delivers it to the federate through the *reflectAttributeValues()* callback.

i. Since there is nothing else for that *tick()* call to do, it returns. The main federate loop finds a flag that a new attribute update has arrived.

j. The loop-back federate (Fed B) performs steps a. to h. to send a reply packet to Fed A.

k. The message arrives at the sender's hardware and steps j. and k. repeat at the original sender's side (Fed A).

l. The callback code samples the receive time (Treceive) and calculates the elapsed_time = (Tsend – Treceive).

m. The one-way latency is: elapsed_time / 2.

n. These events are identical for TCP and UDP except that with TCP more functions are called on each way and the packets have larger headers. In addition, each time a packet arrives, there is an "ack" packet generated and sent in the context of the soft irq. When "ack" is received there is an additional hard interrupt and a soft irq that pre-empts the federate's thread. In this scenario, the federate thread does nothing useful since it is simply waiting for the attribute update message from the far end (Fed B).

5.4.3 Multithreaded RTI, No Computing Load, TCP and UDP

Figure 14 shows a low-level perspective of a multithreaded RTI with no computing load for TCP and UDP. The significant difference in this case is the elimination of the need to

call *tick()* since the RTI now contains sufficient threads to perform asynchronous callback to the federates.

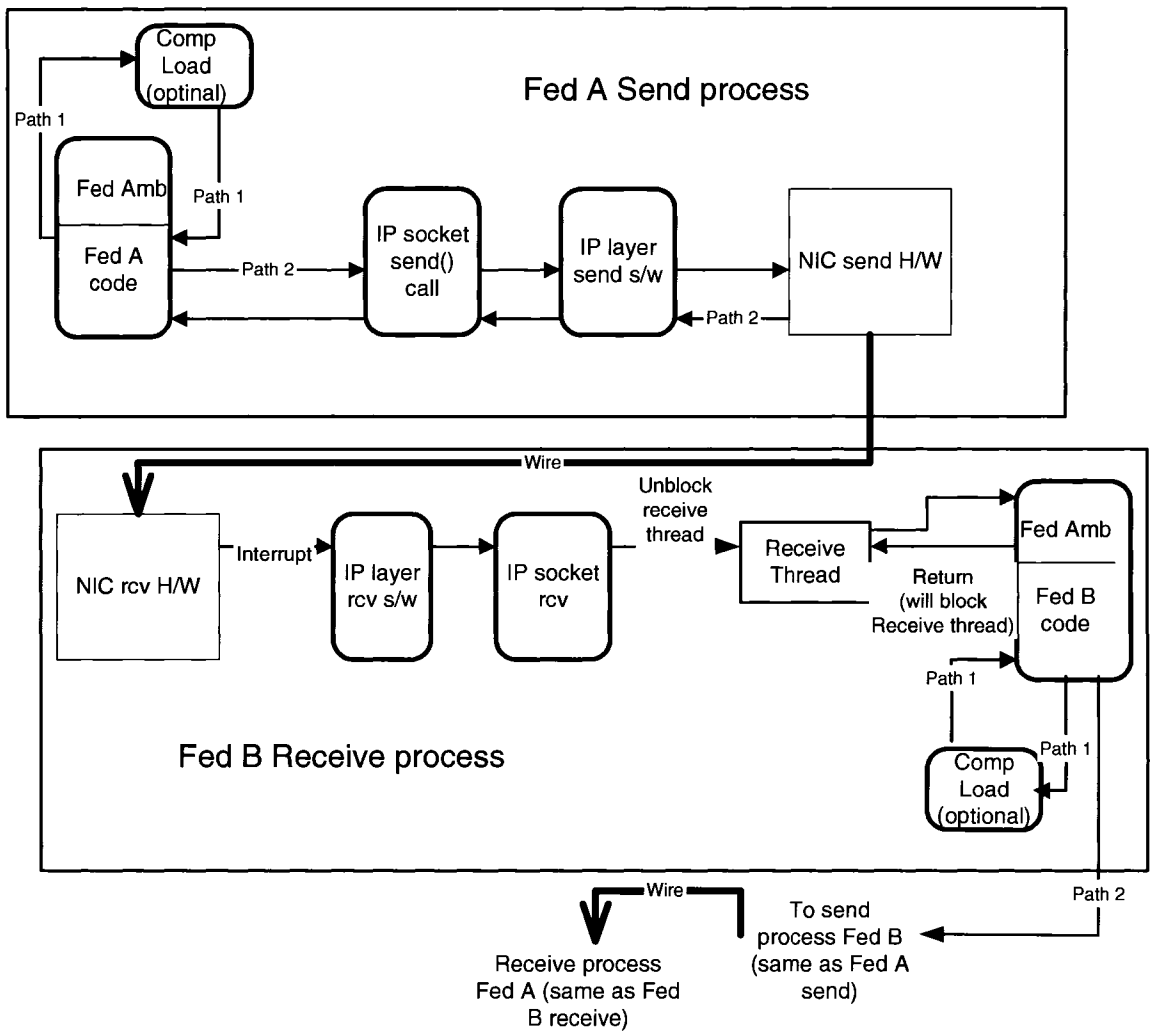


Figure 14–One-Way latency Multithreaded RTI: Low-Level Perspective

In Figure 14, the following events occur:

a. Attribute update message is created by Fed A code.

b. Send time is sampled (Tsend).

c. A call to send *updateAttributeValues()* is generated that invokes socket send and the message starts its travel down the network layers

d. The packet travels in the context of the federate's thread all the way down the IP layers. Since the network device is not busy the packet is copied to the network hardware and sent through the wire

e. The *updateAttributeValues()* function returns and the main federate thread is blocked on an interprocess message queue.

f. Meanwhile, the attribute update travels through the wire, reaches the furthest network hardware, which DMAs it and generates an interrupt.

g. The hard interrupt generates a soft interrupt that delivers the message to the socket to enable interrupts.

h. Since a message was delivered to the receive socket, the Rx high-priority thread that was blocked on the socket until now is unblocked. Through a series of functions it invokes the *reflectAttributeValues()* callback.

i. The *reflectAttributeValues()* callback sends an attribute update message back. The sending all the way down to the hardware is done in the context of the Rx thread (it can be done this way since in the multithreaded RTI federate ambassador methods can invoke RTI ambassador methods without running the risk of recursive function calls).

j. The message travels through the wire back to the sender, and a DMA done interrupt is generated upon packet arrival.

k. Repeat g. and h. at the sender's side.

l. The callback samples the receive time and calculates the elapsed time.

m. The callback, while still in the context of the Rx task, sends an interprocess message (using a system call) to the main federate thread to indicate that the main federate's thread can send the next message. It then returns and blocks again on the receive socket.

n. The main federate thread is rescheduled (unblocked) and is ready to send the next message.

Again the difference in processing between UDP and TCP is identical to the single-threaded case. There will be a small advantage in the multithreaded case since part of the time that *tick()* is processing is saved. There is no difference with respect to interrupts and context switches; after the soft irq, the kernel has to schedule a task anyway and there is no difference between the one that was just pre-empted or one that was blocked on a receive socket earlier.

5.4.4 Single-Threaded RTI, with Computing Load TCP and UDP

Refer back to Figure 12 for a single-threaded RTI, with computing load TCP and UDP.

a. Repeat steps a. to f. in section 5.4.2.

b. Upon return from the *tick()* call, the federate processes internally with standard Drystone cycles; overall duration is: Tdrystone.

c. The loop-back federate (Fed B) has to complete it's processing now (drystone cycles) before it can call *tick()*. Then *tick()* delivers the message to the federate through a callback.

d. The loop-back federate now sends a message back to Fed A.

e. After Fed A finishes internal processing, the Fed A resumes its calls to *tick()*. The incoming message is waiting on the receive socket so *tick()* picks it up and invokes the *reflectAttributeValues()* callback, as in Figure 11.

f. At this stage the roundtrip is complete and round trip time is calculated.

The two-way message latency in this example is approximately Tdrystone since on average both federates perform their computing load concurrently. Most of the message's two-way trip is performed in parallel with internal processing (drystone cycles). Once again, use of TCP would not have much effect on performance.

5.4.5 Multithreaded RTI with Computing Load TCP and UDP

Refer to Figure 13 for the multithreaded RTI with computing load TCP and UDP:

a. Repeat steps a. through g. in section 5.4.3.

b. The *updateAttributeValues()* function returns and the main federate thread processes the computing load as in 5.4.3. It will wait on the interprocess message queue when done.

c. Meanwhile steps i) through m) in section 5.4.3 repeat in the background. The Rx thread pre-empts the main federate thread since it has higher priority and invokes the *reflectAttributeValues()* callback.

d. At this stage the roundtrip is complete and the round trip time is calculated.

e. When the main federate thread is done with the processing of compute load, it does not have to block on the interprocess message queue since the message is already pending on the queue.

Since the RTI is asynchronous and message delivery has priority over the computing load, one- or two-way latency is identical to latency without computing load from internal federates.

5.5 Sender and Receiver Throughput for Single and Multithreaded RTIs – Detailed Analysis

Sections 5.5.1 to 5.5.5 provide a detailed analysis of the sender and receiver throughput experiments for both single and multithreaded RTIs when exchanging attribute updates over TCP and UDP. Section 5.5.6 explains in detail the difference in behavior between single-threaded and multithreaded RTIs when multiple federates are executing on a single processor (CPU).

5.5.1 Sending over UDP Single-Threaded and Multithreaded RTI

When sending an Attribute Update over UDP the following sequence of actions take place in federates code, followed by RTI functions and down to the network layer software and finally the network hardware:

a. The sending federate code creates an Attribute Update.

b. A call to send *updateAttributeValues()* is generated and the message enters the RTI outgoing message queue.

c. If the RTI is single-threaded, it calls *tick()*. Otherwise, go to d.

d. Both single- and multithreaded RTIs loop through all subscribers represented by outgoing sockets and perform the sequence from e. onwards.

e. The packet travels within the federate's thread all the way down the IP layers. If there is room in the send queue of the given socket, the data is copied to the queue.

f. The federate's thread will try calling the NIC transmit function. If the device has enough space in its outgoing buffer, the data goes into the wire.

g. Otherwise (as sometimes happens since the application is faster than the wire), a soft irq is generated and the federate's thread blocks.

h. The *ksoftirqd* thread is scheduled by the kernel and sends all packets, both those in the buffers and in the socket's send queue.

i. When the *ksoftirq* thread is done, the federate's thread is rescheduled to distribute the attribute to other subscribers (federates).

The sender throughput for single and multithreaded RTIs should be almost identical. The limiting factor for large packets is the wire speed.

5.5.2 Sending Over TCP Single- and Multithreaded RTIs

The sequence of operations for sending over TCP single- and multithreaded RTIs is identical to UDP except:

- More work is required to assemble the packet headers.
- The federate's thread may block writing to the socket if the receive side is not ready to accept more packets. This is due to the "sliding window" flow-control mechanism described earlier in section 5.1.2. The key problem with this approach is that the sender will block sending to subscriber A if A is not ready to accept more TCP, even if the next subscriber is ready to accept (subscriber A practically

stalls the process of distribution to the rest of subscribers on the list). This limitation is inherent in the single-threaded process model but is implementation imposed in the case of a multithreaded process model since a multithreaded RTI can schedule additional threads to handle message sending the same way as it uses multiple threads to handle message reception.

5.5.3 Receiving TCP Single-Threaded RTI

The sequence of operations to receive TCP for a single-threaded RTI is as follows:

a. The federate code calls *tick()*, which invokes a sequence of function calls and calls *select()* for each incoming socket. If there is no data on a socket then proceed to the next socket on the list. If there is data on the socket, then invoke handlers and eventually call the callback method. If there is more than one message on a socket then repeat for all messages for this socket.

b. In the background, each incoming packet is processed as described in section 5.1.2, "Message Receive", with the exception that *ksoftirqd* never executes since the main federate's thread never blocks. Thus all incoming packets are handled by a soft irq. This results in relatively long interrupt disable periods for the network hardware interrupt; thanks to the flow-control mechanism, data is not dropped. But the sender federate blocks trying to send until the flow-control mechanism allows it to send. As soon as all incoming packets have been delivered to the receive sockets, the main federate thread is rescheduled by the kernel. The next time *tick()* calls *select()*, a message will be delivered to federates through the federate callback method.

5.5.4 Receiving TCP Multithreaded RTI

When a multithreaded RTI receives Attribute Updates from N sending federates over TCP the following sequence of action takes place:

a. The main federate code calls long *sleep()* and does nothing.

b. N Rx threads block on N receive sockets (recall from Figure 5 that N is the number of peer federates).

c. As soon as incoming packets arrive on the wire, the steps described in section 5.1.2 "Message Receive" are executed. If a packet is delivered to one or more of the receive sockets, an Rx thread is scheduled as soon as the soft irq is complete. The Rx thread delivers all messages by invoking the federate callback method. It blocks when all pending messages have been delivered or when its time slice is over and there is another Rx thread ready.

From the analysis above it is hard to predict which process model should achieve a higher receiver throughput. The single-threaded RTI consumes more CPU and there are many fewer context switches. Interrupts may be disabled for longer periods using single-threaded RTI but this will not necessarily impact overall receiver throughput.

5.5.5 Receiving UDP Single-Threaded and Multithreaded RTI

The receive process for UDP traffic is almost identical to the process for TCP. The absence of a flow-control mechanism implies that some incoming messages may be dropped. The longer periods of disabled interrupts associated with a single-threaded RTI will cause a higher percentage of dropped messages.

5.5.6 Receiving with Multiple Federates on a Single Processor

In large federations it is usually not practical to allocate a processor for each federate. In such a case several federates will have to share the same processor. The process model selected for the federates's RTIs will have an impact on the performance of the federates that share the same processor. The periodic calls to *tick()* that are required in a single-threaded RTI will increase the processor utilization, and higher processor utilization will cause performance to drop.

When using Multiple Single-Threaded Receiver Federates each federate gets 10 milliseconds (a typical OS time slice) of CPU time, even if there are no incoming messages. During the time slice, the federates check for incoming messages by calling *tick()*. While the running federate is polling, other federates on the processor cannot dequeue messages from their sockets and their message queues may overflow. As a result, the TCP flow-control mechanism ("sliding window") causes the sending federate to block. The sender cannot advance to the next attribute update or interaction before distributing the current message to all subscribers. As a result, the stream of messages from a blocked sender stalls until a context switch occurs on the receive side.

With Multiple Multithreaded Receiver Federates All Rx threads of each federate block as soon as there are no more messages available. This allows for more efficient utilization of the processor under all conditions and results in improved receivers throughput. The saving is greater if the incoming packet queue for each receive socket is small. If the queue were large enough to hold 10 milliseconds of incoming packets, there would not be any throughput advantage to the multithreaded process model.

5.6 Theoretical Throughput Gain and Latency Increase with Data-Bundling

Bundling small messages together reduces the number of transmitted and received packets. Bundling results in significant savings in both processing and network resources [12], but introduces additional latency in packet delivery. In RTIs, bundling is performed in the stream manager on a per-destination basis. Messages from clients of the stream manager are combined into larger packets based on the destination. Messages from different sources in the RTI may be combined in the same packet when they are destined for the same address. Upon reception, messages are extracted from the bundled packet and passed to the destination stream manager client. Bundling is controlled by two RID-defined parameters, the bundle size (*MaxBytesBeforeSend*) and the bundle timeout (*MaxTimeBeforeSend*). "Messages are bundled into a packet until either the next message exceeds the remaining size, or the first message placed in the bundle has been delayed for the bundling timeout interval. The packet is then transmitted" [11].

Data-bundling affects throughput in two ways:

a. Bundling of small messages: up to 6 small attribute updates or interactions fit into a single IP packet of standard Maximum Transmit Unit (MTU) size of 1500 bytes on the DMSO RTI. The size of the DMSO RTI message header is 176 bytes [14] and the size of a TCP/IP packet header is 52 bytes. Almost 40 small attribute updates fit into an IP packet when using RTIs such as MAK[8] and FDK[5] that require smaller header sizes. Grouping attribute updates within IP packets improves wire utilization, but a more significant performance gain derives from the fact that it is not necessary

to generate a hard interrupt + soft interrupt + potential context switch for each RTI message (with a performance cost that can be as high as 2 milliseconds [21]).

b. Reduction in amount of hardware interrupts: there is improved data throughput for large data buffers, and even for data buffers that exceed the MTU, because the IP packets are sent and received one immediately after the other. A single hard interrupt, soft interrupt and thread switch (from the application thread to the softirqd thread) occurs for several packets instead of for a single packet. For example, for a data-bundling configuration parameter *MaxTimeBeforeSend* with a value of 0.02 seconds and using data size of 4096 bytes with MAK[8] RTI, about 57 RTI messages are bundled together. Up to 56 hard interrupts and context switches can be saved (at the receive side). In this example the data throughput improves by about 28%. The use of bundling may also introduce latency, assigning *MaxTimeBeforeSend* to a value of 0.02 seconds for example, will cause a worst-case message latency of 0.02 seconds for the first message in a bundle. Subsequent messages added to the bundle will suffer less latency, and a message added just before the bundle is transmitted will not suffer any additional latency due to the *MaxTimeBeforeSend.*

5.7 Potential Performance Gain Using Unreliable Transport (UDP)

UDP may outperform TCP due to the following reasons:

- UDP packet headers are 8 bytes shorter than TCP packet headers. Assuming an RTI header of 16 bytes (such as the FDK [5]) and an attribute update or interaction of the same size, network traffic can be reduced by 20%. This in turn reduces time through the wire. With powerful hardware (short interrupt response time and irq handling)

and a relatively slow network (100Mbit/sec or less), the increase in throughput comes close to the reduction in time through the wire.

- The construction of UDP packets requires fewer steps and processing effort, but using CPUs with a speed over 1GHz and RAM in excess of 64Mb, the gain is minor.
- The TCP protocol requires acknowledge packets to be returned to the sender. When the sender of an RTI transaction receives a TCP "ack" packet, it has to deal with extra workload as when receiving any other IP traffic (for example, a hard interrupt or soft irq). As explained above, this may amount to an extra 100 usec per payload packet. On a half-duplex network infrastructure the ack packet may consume valuable bandwidth. On full-duplex infrastructure the ack packet travels against the busy direction of the data so the extra workload is not significant.
- Additional network load does not affect the sender throughput. It does however reduce the number of data packets that reach the receiver and therefore reduces its measured throughput (assuming that throughput is measured at the receivers side).
- Additional network load increases the number of collisions and forces retries. With TCP, retries are handled by the IP layer but with UDP they are handled by the federate code, which must increase the amount of send cycles to compensate for lost transactions. Since the IP layer deals with lost packets in a more efficient way than federate code, TCP data throughput should be less affected. In the latency test, flow control is inherent in the tests algorithm (the return message acts as an ack), so the TCP ack packet is redundant and negatively affects performance.

5.8 Algorithm and Calculation for Time-Advance Grant

The main cause of reduced performance using a time-managed federation is the costly overhead. In a time-managed federation, simulation steps take place at each point on the logical time scale. In order for the federation to advance to the next step, the federation's time must advance through a request initiated by the federate and a grant given by the RTI. Computation of the next granted time requires calculation of the Lower Bound on Time Stamp (**LBTS**). The following considerations should be made with respect to the LBTS:

- The LBTS value computed for a federate is the lowest (earliest) time stamp of messages that are destined for that federate later in execution;
- For each federate, the RTI must ensure that:
  - Time Stamp Ordered (**TSO**) messages are delivered to the federate in time-stamped order, and
  - No message is delivered to the federate with a time stamp that is earlier than its logical time.
- Once the LBTS for a federate is computed:
  - The RTI delivers to the federate all TSO messages with a time stamp earlier than LBTS, and
  - If the RTI prevents the federate from advancing its logical time beyond LBTS, it guarantees that the federate receives no messages in its past.

- To compute the LBTS, the RTI must consider:
  - The earliest time stamp of a TSO message that any federate might generate in the future (the current logical time of a federate is one boundary since no federate can generate a TSO message in its past), and
  - The time stamps of messages within the RTI and the interconnection network (transient messages).

All RTIs use the butterfly scheme for LBTS computation. This scheme is similar to well-known parallel prefix and barrier algorithms [19]. A single LBTS computation is initiated when a federate requests a time advance or next event and requires NlogN messages to be exchanged between the federates [19] (where N is the sum of time-regulating and time-constrained federates in the federation). RTIs deal with transient messages (messages that have been sent, but have not yet been received) in different ways.

Actual LBTS calculations may require between Nlog N and (T+N)log N messages [12] (where T is the number of transient messages in the system when the LBTS calculation is initiated) were each message contributes it's transfer time between the sending and receiving federate (message latency) to the overall LBTS calculation. Some of these messages can be processed concurrently but a significant degree of serialization will always be present, proportional to the depth of a binary tree with a federate at each node. For example, in a federation of 8 federates that are all time-regulating and time-constrained, and which are interconnected over a WAN with average network latency of 0.1 seconds, the minimal delay for a federate to advance to the next point in time is: log 8

* 0.1_{sec} = 0.3 seconds (the amount of serialization is log N not Nlog N). Such a delay is not acceptable for a human-in-the-loop simulation.

Chapter 6: Measured Results

The hardware setup shown in Figure 15 was used during all test runs.

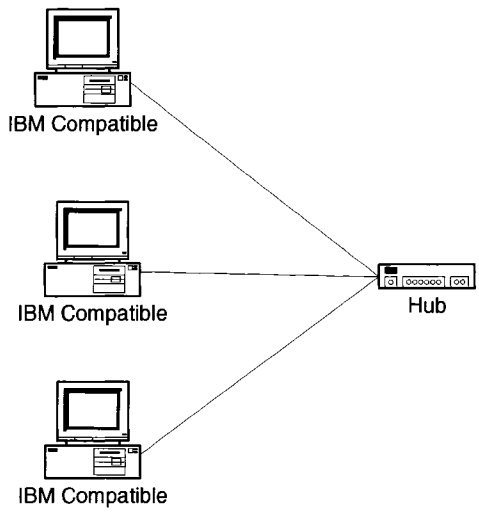


Figure 15–Hardware Used for Testing

All the PCs had a 900 MHz single Athlon processor, 192 Mbyte of RAM, and were equipped with a PCI 100 Mbit/sec Ethernet interface interconnected through a 100 BaseT hub. Linux Red Hat 6.2 (Kernel version 2.2.14) was used since the FDK version 4.0 compiles only under this Linux kernel. The DMSO RTI version 1.3 NGv4 was used for the data-bundling and time-management experiments. The FDK version 4.0 was used for the other experiments.

Attribute updates were used to collect performance data for the first three enhancement techniques: multithreading, data-bundling and the use of optimal IP protocols. It was discovered than there is no difference in performance between the use of attribute updates or interactions. There is only a distinction at a higher level of RTI software, which doesn't affect performance. There was no evident affect on performance with both the

FDK and DMSO RTIs. Since attribute updates are more commonly used in simulations, they were used throughout all the experiments as the basic message-exchange unit between federates. All attribute updates had the following structure:

```
{
    int index;
    int data_length;
    char message[data_length];
}
```

Data lengths of 32 to 2048 bytes were used to experiment with data sizes greater than the MTU. The MTU was set to a typical size of 1500 bytes. This implies that effective packet size (payload) was 1464[24] bytes over UDP and 1448 over TCP[24] (These numbers are for Berkeley Software Design (BSD) implementation used by the Red Hat Linux).

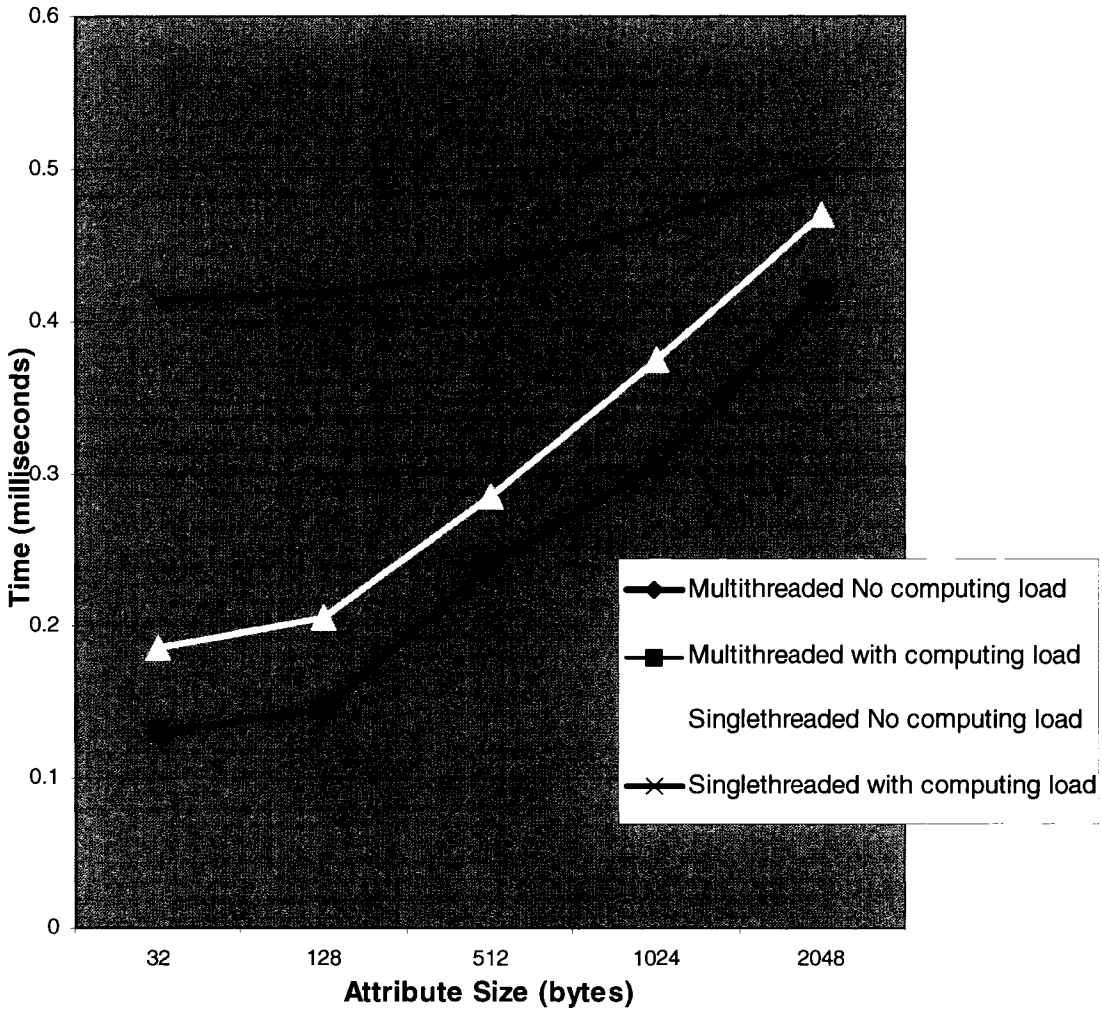
The experimental results had a variance of less than 5%, due to occasional timing misalignments between messages at the receiving node. Timing misalignments affect the queuing of messages and may slightly and occasionally affect the efficiency of data movement through the layers of the protocols (IP and RTI). As more federates are involved, the more slight changes in timing will affect performance and generate variance in results. StartTime and endTime were obtained using the Linux system call *getTimeOfDay()* that is described and analyzed in detail in Appendix B.1

6.1 Effects of Multithreading

The following comparisons were made between a single-threaded and a multithreaded RTI, both based on the FDK version 4.0:

- One-way latency with and without computing load
- Receiver throughput
- Sender throughput
- Concurrent send and receive

The One-way latency experiment was repeated for 10 test runs for each attribute (message) size and the mean latency was calculated from 1000 one-way latencies measurements, as shown earlier in Figure 6. Finally, the average of the 10 test runs was calculated for each data size.



Error! Bookmark not defined.

Figure 16–Single- vs. Multithreaded: One-Way Message Latency and Computing Load

Figure 16 presents a comparison of one-way message latency for two federates with and without computing load as described in detail in section 5.3.1 (note that the line for multithreaded with computing load overlaps with the line for M\multithreaded without computing load). Each data point on the graph represent the average of the ten test runs that were performed for each data size. In each test run 1000 attribute updates were sent and then received by the initiating federate. The variance in measured latency between

test runs was less than 1%. Computing load was generated by the federate's code, which had to perform 1000 Drystone cycles (0.6 milliseconds processing time) after each attribute update was sent to the second federate. A value of 0.6 milliseconds was chosen as the compute load since it is of the same magnitude as the one-way message latency. The results show that without additional computing for the federate's thread, the one-way latency is shorter when a multithreaded RTI is used for all data sizes. This result is explained by the fact that the message is presented to the federate ambassador immediately rather than having to wait for a "check-whether-a-new-message-is-available" request (*tick()*) from the federate's code. Additional computing load doesn't affect the responsiveness of the multithreaded RTI since the receive thread has higher priority than the main federate's thread (the thread that executes the computing load). On the other hand, the single-threaded RTI responds more slowly to incoming data since it is executing the computing load. The graph shows that even though the single-threaded RTI was executing a 0.6 millisecond computing load per cycle, the effect on measured latency was on average 0.25 milliseconds or less. This result is explained by the fact that the return message can arrive at any time during the execution of the computing load. On average, *tick()* was called sooner than 0.6 milliseconds after the message returned. The one-way latency for a single-threaded RTI without compute load is about 20% lower than was recorded by Fitzgibbons [11] who used the same RTI. The difference can be attributed to more powerful hardware (a dual-Pentium II 300-Mhz PC compared to the single 900-Mhz Athlon PC used in these tests).

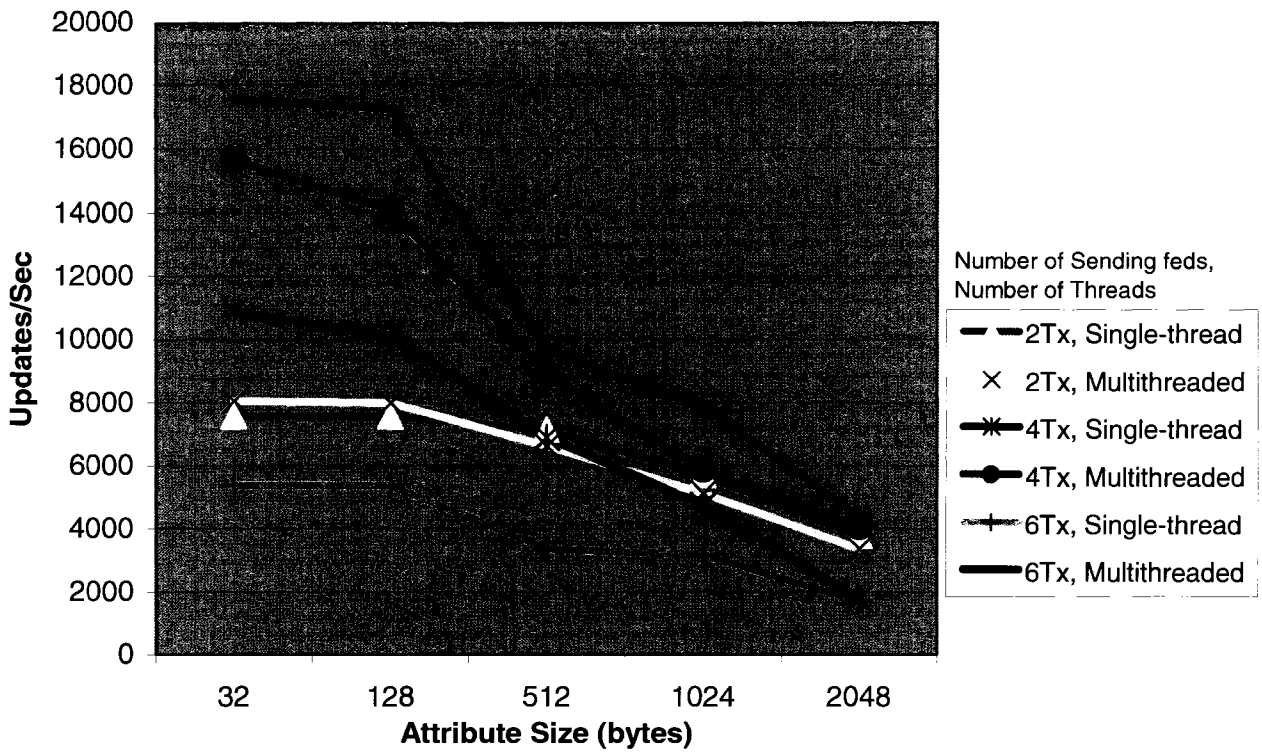


Figure 17–Single-Threaded vs. Multithreaded RTI: Average Receiver Throughput

In the average receivers throughput experiment described in section 5.3.3, all senders were multithreaded in order to isolate the effect of multithreading on receiver federates. Ten test runs where performed for each data size and for each quantity of senders. Each data point in Figure 16 represents the average results of the 10 test runs. In each run, 10,000 attribute updates reached the receiving federate, meaning each of the N sending federates sent 10,000/N messages. The variance in throughput between test runs was less than 5%.

Figure 17 shows that multithreaded RTIs provide higher receiver throughput. It also shows that a single federate acting as a receiver can handle more incoming messages than a single sending federate can distribute. While the multithreaded RTI was able to handle

data from at least 6 senders concurrently, the single-threaded RTI slowed down when higher volumes were incoming. This demonstrates the advantage of interrupt-driven receive operations performed by network hardware and software as used in the multithreaded RTI over the polling-driven receive operations done by the application when using a single-threaded RTI.

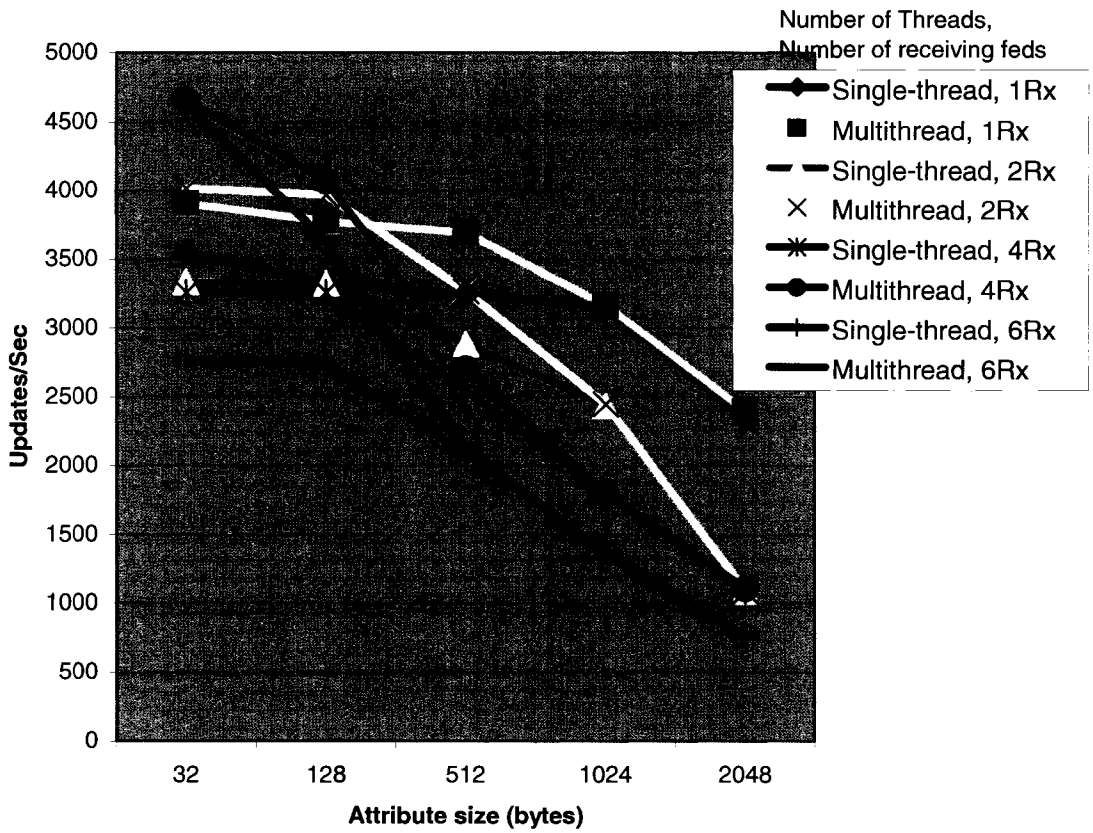


Figure 18-Single-Thread vs. Multithread: Average Sender Throughput

The sender throughput experiment was performed with multithreaded receive federates. As explained in section 5.5.6, it was discovered that when more than a one single-threaded federate executes on a machine, performance drops dramatically and processor utilization increases. The built-in Linux "System Monitor" reported a total CPU utilization greater than 85% when attempting to execute 3 single-threaded receive

federates concurrently, while the CPU utilization with 3 concurrent multithreaded receive federates was represented as less than 35%.

The sender throughput test procedure was described in chapter 5.3.2. Ten test runs were performed for each data size and for each quantity of receivers. Each data point in Figure 18 represents the average results of the 10 runs. In each run, 10,000 attribute updates were sent by the sending federate to each receiving federate. The variance in measured throughput between runs was less than 5%. The results did not demonstrate a noticeable performance difference between the two process models when using message sizes of 512 bytes or greater, but the multithreaded RTI performed about 25% better for smaller message sizes. Since send operations occur in the federate code; multithreading does not come into play. The fact that a *tick()* call is not required in the multithreaded RTI in order to send data does improve performance moderately, but only for small messages. For larger messages the network layer limits the throughput regardless of process model. Note that the DMSO RTI doesn't require a call to *tick()* in order to perform send operations from within the federate code.

As figures 17 and 18 demonstrate, a single federate can handle concurrent incoming messages from several senders since the federate's receive throughput is greater than the federate's send throughput.

This conclusion conflicts with the results published by Fitzgibbons [11], who used the same RTI and operating system. They show a greater or equal sender throughput than receiver throughput for any quantity of federates. Fitzgibbons [11] does not explain the purpose of using 2, 4 or 8 federates. It may be that during both the receiver and sender throughput experiments, Fitzgibbons [11] used a single sender and several receivers.

Knight [12] doesn't mention the number of federates in their sender and receiver throughput experiments but their results suggest that at least for smaller messages, the receiver throughput exceeds sender throughput (though only by 20%). Knight [16] benchmarked four RTIs for throughput including the FDK and DMSO NG1.3v3.2. Though they state that they used an identical number of senders and receivers, their results show a much greater sender throughput than receiver throughput for the same RTI. For example, their sender throughput for the FDK appears to be 8 times the receiver throughput, while for the DMSO RTI it is 20% better than receiver throughput for small data sizes and identical for larger data. Since Knight used an identical number of sending and receiving federates it is unclear how their receiver throughput can differ from sender throughput unless they used such a short number of transactions that the data could have been queued somewhere.

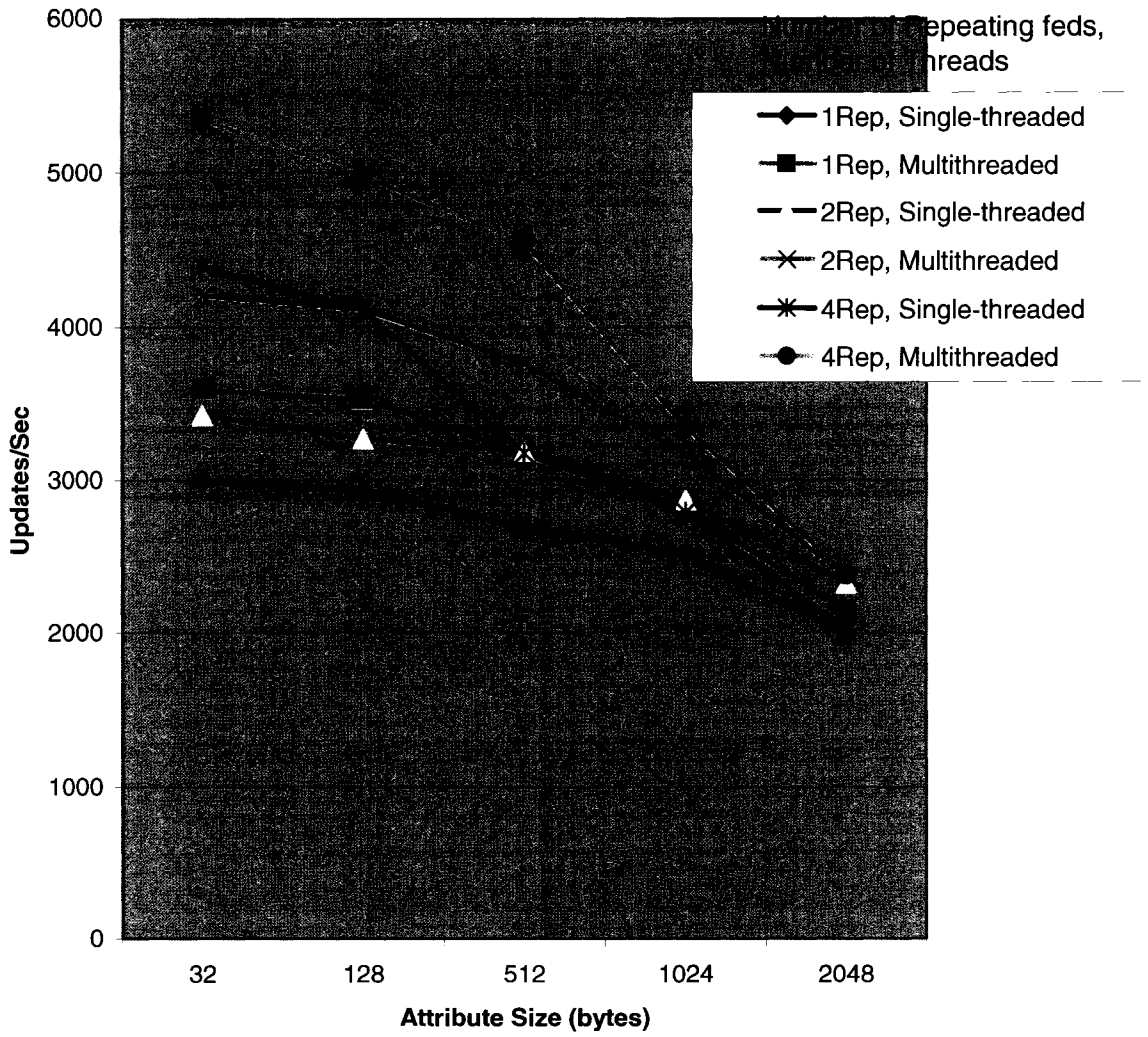


Figure 19- Single-Threaded vs. Multithreaded RTI: Average Loop-Back Throughput (concurrent send and receive)

For the experiment that generated the results in Figure 19, a single "master" federate sent and received concurrently between 1 and 4 repeaters or "slaves". All measurements were taken at the master. The repeater federates were multithreaded since, as explained in section 5.5.6, performance drops dramatically for more than one single-threaded federate per processor.

The test procedure is described in section 5.3.4. Ten test runs were performed for each data size and for each quantity of receivers. Each data point in Figure 18 represents the average results of the 10 runs. The variance in throughput between test runs was less than 5%. The master sent a total of 10,000 attribute update messages. With N repeaters, each repeater received and returned 10,000/N messages.

Throughput with more repeaters is usually better. With larger data sizes, throughput became limited by the bandwidth of the wire. More useful results were measured with smaller data sizes, of between 32 and 512 bytes. Figure 18 shows that the throughput advantage of the multithreaded master becomes more marked with more repeaters and larger data sizes until it levels off due to wire capacity.

Preliminary tests with 8 repeaters confirmed that the multithreaded master achieved better throughput than with 4 repeaters, but the single-threaded master slowed down. With 8 repeaters (a total of 9 federates), the RTI occasionally froze, however, and was unsuitable for experimentation and data collection.

6.2 Effects of Data-Bundling

The Maximum-Time-Before-Send parameter (*MaxTimeBeforeSend*) is the time period during which packets are bundled at the sender before being distributed to receivers. It must be noted that the DMSO NG1.3v4 RTI didn't respond to the second data bundling parameter that is the *MaxBytesBeforeSend*. The DFK version 4.0 doesn't support data bundling.

The average sender throughput was measured between a single sender and a single receiver using the procedure described in section 5.3.2. The DMSO RTI NG1.3v4 was

used for this experiment. Ten test runs were performed for each data size with 10,000 attribute updates in each run. Each data point in Figure 20 represents the average results of the 10 test runs.

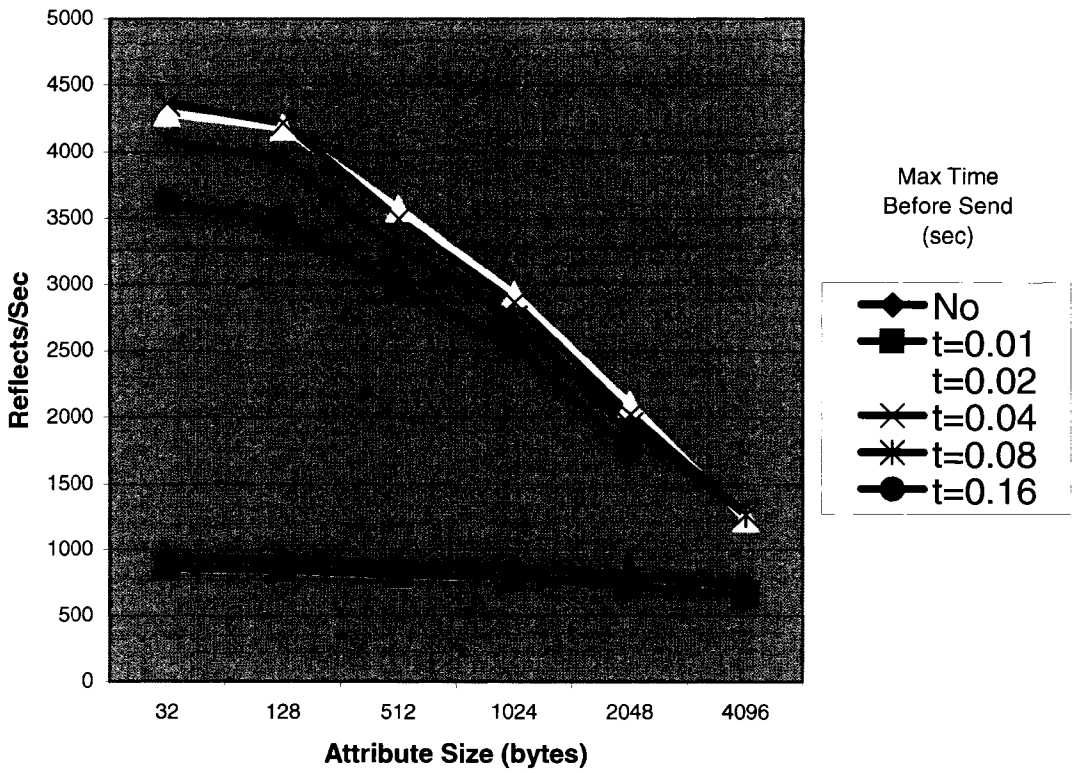


Figure 20–Data-bundling - Average Sender Throughput: Finding the optimal *MaxTimeBeforeSend*

The variance in throughput was less than 1%. A bundling time of 10 ms didn't improve performance and there was a small negative effect due to additional data-buffering. The best results where achieved using buffering times of 20ms and 40ms. Higher values (80ms, 160ms) decreased performance.

These results can be attributed to unbalanced bursts of data load. Figure 19 also shows that at very large data packets (4096-bytes with a value of 160 milliseconds for *MaxTimeBeforeSnd)* the network layer actually dropped data. After making this

discovery, *MaxTimeBeforeSnd* was reduced to 20 milliseconds to measure the effects of data-bundling on performance.

Average sender throughput was measured using the procedure described in section 4.3.2 for a single sender and 1, 2 and 4 receivers. Ten test runs where performed for each data size with 10,000 attribute updates in each run. Each data point in Figure 21 represents the average of the 10 runs. The variance in throughput between test runs was less than 1%.

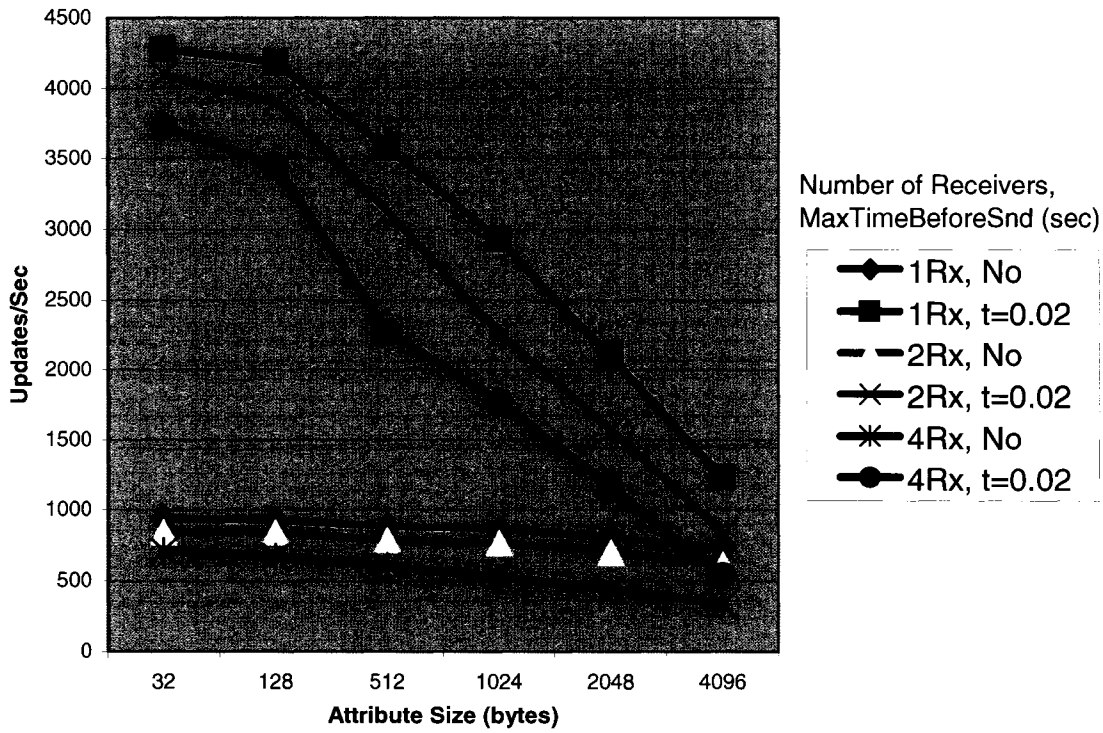


Figure 21- Data-Bundling: Average Sender throughput for the DMSO RTI NG1.3v4

Figure 21 shows that data-bundling significantly improved the sender throughput for all tested quantities of receivers, especially for smaller data sizes. Data-bundling is less effective for larger data sizes since the number of sent IP packets is not reduced by as

much. At large data volumes (large messages and many recipients), data-bundling is more effective since the limiting factor becomes the bandwidth limit of the hardware layer. For small messages, utilization of the hardware layer was less than 0.1 with bundling and less than 0.02 without. (Refer to appendix B.2 for calculations of hardware-layer utilization.) Figure 21 also shows that even when message size exceeds the MTU, throughput is 250% higher with data-bundling.

Average receiver throughput was measured using the procedure described in section 5.3.3. Ten test runs were performed for each data size and for each quantity of senders. In each run 10,000 attribute updates were sent to the receiving federate, implying that each of N sending federates sent 10,000/N messages. Each data point in Figure 22 represents the average throughput of the 10 runs. The variance in throughput was less than 5%.

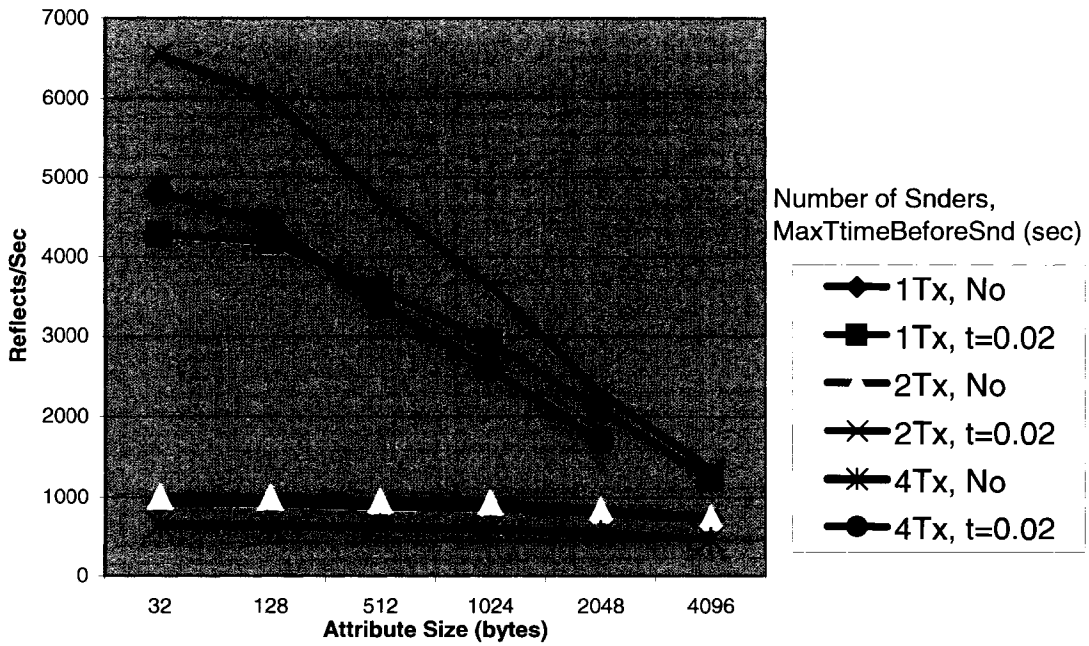


Figure 22–Data-Bundling: Average Receiver Throughput for the DMSO RTI NG1.3v4

Figure 22 shows that both with and without data-bundling, the best receiver throughput was achieved with two sending federates. When sending small attributes, the receiving federate was underutilized, especially with data-bundling. With 4 senders and attribute sizes of 4096 bytes, the receiver failed to keep up with incoming data and dropped packets. At an attribute size of approximately 512 bytes, the receiver's performance with 4 senders started to drop below its performance with a single sender. This can probably be attributed to the fact that the network layer at the receivers side handles data flow from 4 separate sockets (there is a socket allocated for each federate) in a less efficient way than when there is only a single receiving socket and hence, performance degraded under higher data rates (around 25 Mbit/sec). In the worst-case, data-bundling increased latency by the value of *MaxTimeBeforeSend.* It must be noted that the DMSO NG1.3v4 RTI didn't respond to the second data bundling parameter that is the *MaxBytesBeforeSend.* The DFK version 4.0 doesn't support data bundling.

6.3 Connection-Oriented vs. Connectionless Data Exchange

Commercially available RTIs such as DMSO and MAK implement the HLA specification. This requires support of "reliable" and "best effort" data exchange for both interactions and attribute updates. The .fed file defines which interactions and attribute updates will be exchanged as "best effort" and which will be exchanged as "reliable". connection-oriented IP (TCP) is used for data that requires "reliable" delivery and connectionless IP (UDP) for data that the user designates for "best effort" delivery. Users often assume that if they define an interaction or attribute update as "best effort", they will reduce network load and improve overall performance even if they lose some

transactions. Figures 23 and 24 demonstrate the difference in latency and data throughput between connection-oriented and connectionless data exchange.

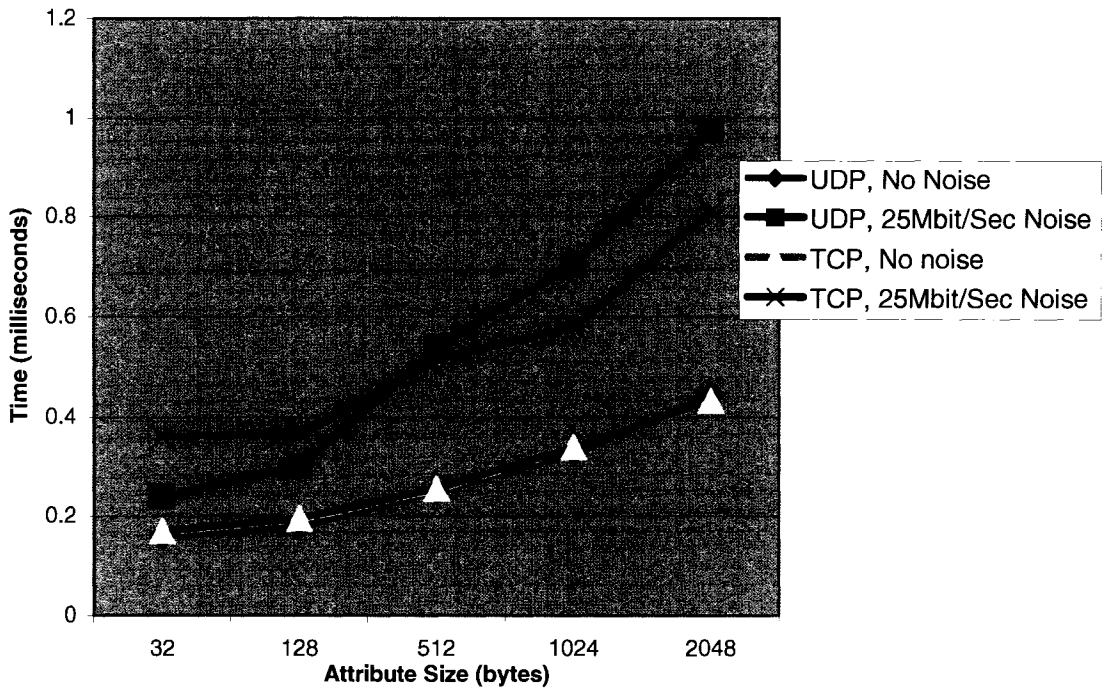


Figure 23–TCP vs. UDP: Average One-Way Latency for the FDK version 4.0

The one-way latency test is described in detail in section 5.3.1. Ten test runs were performed for each data size. In each run 1000 attribute updates were sent and then received by the initiating federate. Each data point in Figure 23 represents the average results of the 10 runs. The variance in throughput between runs was less than 2%. An additional machine distributed an additional network load of 25 Mbit/sec over the LAN to simulate shared network conditions with other applications. Figure 23 shows that when no additional network load was present, there wasn't a significant difference in message latency between TCP and UDP. When the additional network load was

introduced, there was an advantage to UDP for messages smaller than 512 bytes. For larger messages (higher network load), TCP performed better. In both cases (with and without network load) there was no message loss when using UDP. Knight [12] and Lorenzo [25] observed a slight improvement in message latency when using TCP over LAN with the DMSO RTI; however, they didn't test latency under additional network load.

The average receiver throughput test was performed between a single sending and a single receiving federate according to the procedure described in section 4.3.3. For each IP protocol, 10 test runs were performed for each data size. In each run 10,000 attribute updates were received by the federate (note that if messages are being lost, the senders will send a total of more then 10,000 attribute updates). Each data point in Figure 24 represents the average of the 10 runs. The variance in throughput between test runs was less than 1%.

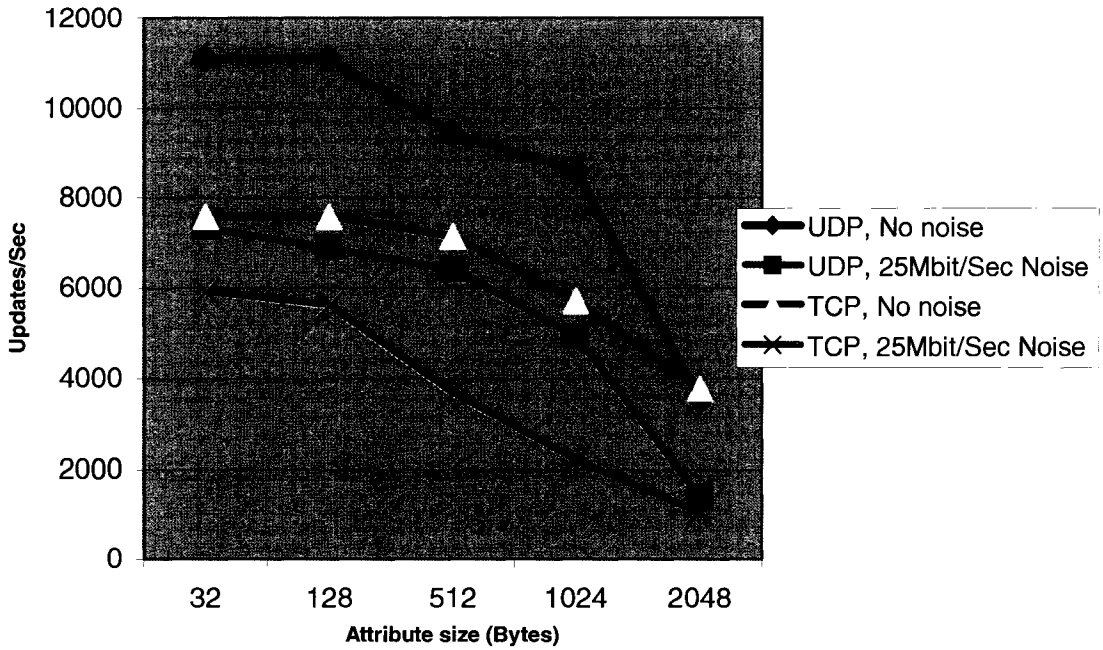


Figure 24–TCP vs. UDP: Average Receiver Throughput FDK version 4.0

Figure 24 shows the throughput advantage of UDP over TCP. This advantage is due to less demanding processing by the network layer and has nothing to do with the RTI. When the network was loaded, a significant percentage of attribute updates were lost with UDP. With larger messages, the retries were handled by the application rather than by TCP layer.

Knight [12] observed a 20% throughput advantage to TCP over UDP with MAK RTI and 15% advantage to TCP over UDP with DMSO NG1.3v4 RTI.

6.3 Performance Cost of Time Management

Since the results for the time-advance grant benchmark published by Fitzgibbons [11] didn't agree with the theory, the same test procedure was repeated. An unmodified DMSO BmTimeAdv federate code was used on more powerful hardware (a single 900-

Mhz Athlon PC instead of a dual Pentium II 300-Mhz PC), and in both cases the OS in both cases was RedHat Linux and a 100-Mb Ethernet network was used. The test procedure for time-advance grant throughput is described in section 5.3.5. The test was carried out for 2, 4, 6 and 8 federates, all time-regulating and time-constrained. The test was executed for 10 runs. In each run each federate generated 1000 time-advance requests. In all tests, a default look-ahead value of one unit of simulation time was used. Experiments were done using larger values, but they didn't lead to a noticeable change in performance.

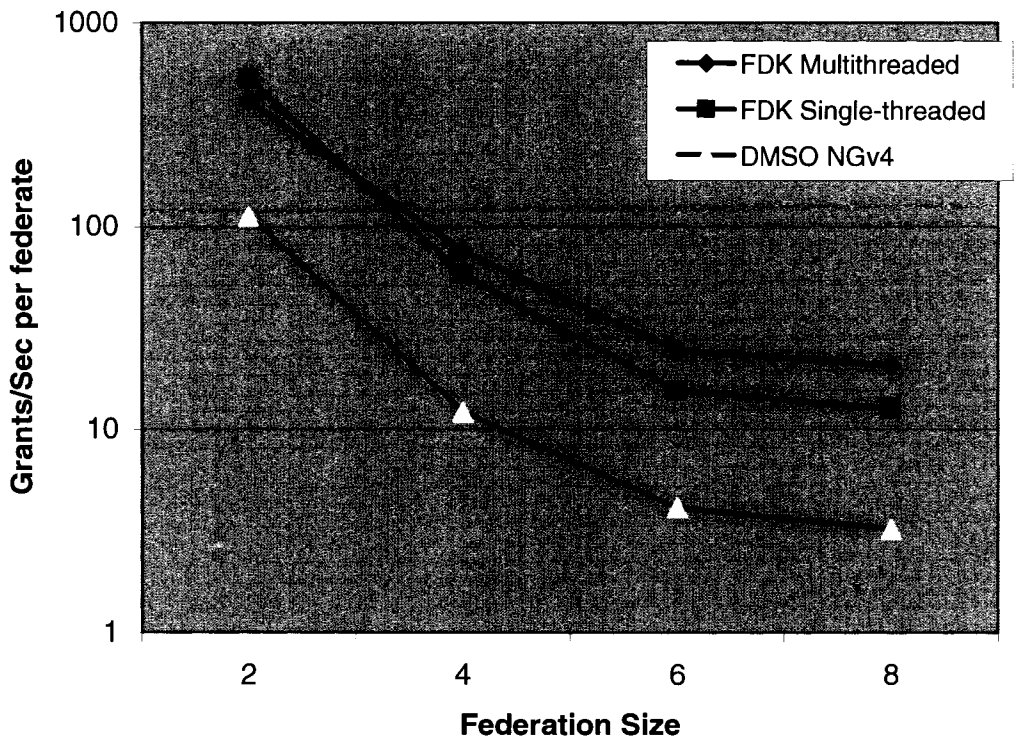


Figure 25–Time Advance Grant Throughput over LAN for Various RTIs

Figure 25 presents the results obtained over a LAN, as was done by Fitzgibbons [11]. Fitzgibbons in his research used a version of the FDK that is identical to the unmodified single-threaded FDK used in this research. The one way latency for that RTI was about 20% faster in this research than was measured by Fitzgibbons (probably due to more powerful hardware). Fitzgibbons included his one way latency results in the same paper, and presents a Time Advance Grant performance that is at least 1000% higher than measured in this research.

The results presented here show 14 time-advance-grants/second using the single-threaded FDK with 8 federates that are all requesting time advance grants concurrently. The one-way message latency measured for that RTI is 0.3 milliseconds. Without accounting for transient messages there are 8 * 8 * log 8 = 192 messages exchanged before all 8 federates have there LBTSs calculated. At one-way message latency of 0.0003 seconds there are 1/(192 * 0.0003) = 17.36 time advance grants expected each second. The much higher one-way message latency of the DMSO NG1.3v4 RTI leads to 3.2 time advance grants/second using that RTI. The results are consistent with theory. The cost of LBTS calculation is high. For example, with 6 federates or more, delays due to LBTS calculation increase response time beyond 50 or 100 milliseconds (depending on the RTI) and a "human-in-the-loop" simulation becomes impractical since according to Merikle [26], real-time interactions with a human in the loop require response times better than 100 milliseconds.

6.5 Unexpected Results

The following results were unexpected:

- Figure 17 shows that the single-threaded RTI with 4 sending federates demonstrated a higher receiver throughput than with 6 sending federates. This is unexpected because the receiver federate was the only active user thread on its processor during these runs, and the polling mode should have been more efficient with respect to CPU usage. One possible explanation is that the polling rate of the federate was too high and caused partial starvation to the network layer (i.e. the *ksoftirqd thread*). These results support Karlssons [6] hypothesis that a single-threaded RTI requires careful tuning of the federate to prevent starvation of the network layer, while a multithreaded RTI does not.
- Figure 23 shows that while UDP in noisy environments demonstrated lower message latencies for smaller messages, it showed higher message latencies than TCP for larger messages. This is unexpected because no messages were lost with UDP regardless of size, and under this conditions, UDP should be expected to provide better performance for any given message size. Lorenzo [25] observed a slight improvement in message latency when using TCP over LAN with the DMSO RTI. Knight [12] discovered the same behavior though without additional network load/noise. Knight doesn't suggest an explanation; however, this behavior should be attributed to the network layer and not to the RTI. The RTI layer is unaware of data transport details and uses the socket layer rather than directly accessing the lower TCP/UDP layer. Both Lorenzo and Knight used the Linux OS (and it's IP implementation) as was done in this research. The result may be different with a different IP implementation.

- Figure 18 shows that when using a multithreaded sender and small messages (32-128 bytes), throughput per receiver improved slightly with additional receiving federates. This is unexpected since multicasting was not used, and the increase in receiving federates implies more outgoing messages for the sender and hence reduced throughput per receiver. This result should be attributed to timing alignments during the test runs that increased the efficiency of data transfer.

Chapter 7: Conclusions and Summary of Contributions

In this research multithreading, data bundling and selection of IP protocol were investigated as means of enhancing performance of RTI. In addition the performance cost of time management was measured. This research also proposed few improvements to the current RTI performance measurement techniques.

The following conclusions are derived from this research:

1. As shown in section 6.1, the one-way Latency with and without computing load for the multithreaded asynchronous RTI is more responsive (i.e. has improved message

latency) when the federate code is performing work in addition to receiving messages.

2. Since polling (*tick()*) is eliminated in the multithreaded asynchronous RTI, processor utilization is generally reduced, enabling more federates on a single processor while still providing high responsiveness for each federate. This was discovered during the original "Sender Throughput" experiment (section 6.1) when using multiple single-threaded receivers on the same machine, there was higher processor usage as was indicated by the OS performance monitor.
3. Reduced processor utilization improves receiver throughput especially when several federates are sharing a single processor. This was demonstrated by the "Receiver Throughput" experiment (section 6.1).
4. As shown by the "Sender Throughput" experiment (section 6.1), multithreaded asynchronous RTIs have only a minor throughput advantage when sending data.
5. As shown by the "Concurrent Send and Receive" experiment (section 6.1), in practical scenarios in which a federate has to send and receive concurrently, the multithreaded architecture performs at least 20% better.
6. Data-bundling improves throughput but increases worst case latency to at least *MaxTimeBeforeSnd.*
7. As shown by the "Average Sender Throughput: Finding the optimal MaxTimeBeforeSnd" experiment (section 6.2) there is a specific range for *MaxTimeBeforeSnd* that provides best throughput for a given network and OS combination.

8. As shown in the "Data-Bundling Throughput" experiments (section 6.2), data-bundling improves throughput even when using messages larger than the MTU.
9. When comparing RTIs with different architectures, testing with concurrent input and output is essential.
10. As shown in Figure 22, use of reliable transfer (TCP) provides lower message latency when significant network traffic is present and larger messages are being exchanged.
11. As shown in Figure 16 the impact of additional computing load depends on the process model. The multithreaded process model is more resilient to the addition of computing load.
12. HLA time management should be utilized carefully in real-time simulations. The large overheads associated with time management may slow the simulation unacceptably, particularly for human-in-the-loop simulation.

This research:

1. Demonstrated the performance advantages of multithreaded asynchronous RTI.
2. Explained and demonstrated data-bundling as a way to increase message throughput.
3. Demonstrated the different performance of reliable (TCP) and unreliable (UDP) data transfer in the presence of additional network traffic.
4. Introduced an additional RTI benchmark that models "real-life" conditions by sending and receiving concurrently.
5. Introduced computing load and network load to standard HLA benchmarking.

6. Proved that for real-time federations with 8 or more federates, time management may be impractical.

Future Research:

There is an increasing demand in the defence community for simulations with federates in several geographical locations. A distributed simulation over a WAN presents several challenges due to higher latencies and usually lower available bandwidth. It will be particularly interesting to look at performance of time managed federations over a WAN, and at alternatives to time managed federation such as "real-time" federations based on time synchronization (through GPS clocks or NTP). In order to address the issue of low network band-width, the total amount of HLA traffic can be potentially optimized using Data Distribution Management (DDM), which is part of the HLA standard but rarely used due to lack of information, or through extensive use of multicasting. These two techniques should be further tested to develop guidelines for federation developers.

References

[1] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Framework and Rules. Last obtained Jan 18, 2006 from:
http://standards.ieee.org/catalog/olis/compsim.html

[2] R.Drake, P.Agarwal, M.Espinosa, C.Owens, R.Liebel, J.Steele, "Independent Benchmarking of RTI Real-Time Performance". COLSA Corporation. Fall 2003 Simulation Interoperability Workshop (SIW).

[3] J.O.Calvin, R.Witherly, "An introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)". The MITRE Corporation. Last obtained Nov 22, 2005 from: http://dss.ll.mit.edu/dss.web/96.14.103.RTI.Introduction.html

[4] Department of Defence, Defence Modeling and Simulation Office, "RTI 1.3-Next Generation Programmer's Guide" Version 6. Was available from http://www.dmso.mil

[5] Georgia Institute of Technology. Federated Simulation Development Kit (FDK) Version 4.0. (2002). Last obtained Jan 25, 2005 from: http://www.ece.gatech.edu/

[6] M. Karlsson and P. Karlesson, "An In-Depth Look at RTI Process Models". Pitch AB. Spring 2003 Simulation Interoperability Workshop (SIW).

[7] L. Granowetter, D.D.Wood, "Opening the Black Box: The Effect of RTI Implementation Details on the Success of an HLA Federation". Spring 2003 Simulation Interoperability Workshop (SIW)

[8] MAK High Performance RTI. Last obtained Nov 22, 2005 from: http://www.mak.com/rti.php

[9] Pitch's portable Runtime Infrastructure. Last obtained Nov 22, 2005 from: http://www.pitch.se/hla/default.asp

[10] Department of Defence, Defence Modeling and Simulation Office, "BmLatency benchmark source code" Was available from http://www.dmso.mil

[11] B.Fitzgibbons, T.McLean, R.Fujimoto, "RTI Benchmark Studies". Spring 2002 SIW

[12] P.Knight, R.Liedel, M.Klinner, J.Steele, "Analysis of Independent Throughput and Latency Benchmarks for Multiple RTI Implementations", Fall 2002 Simulation Interoperability Workshop (SIW).

[13] T. Burks, T Alexander, K. Lessmann, "Latency Performance of Various HLA implementations". Amtec Corporation. Last obtained Nov 22, 2005 from http://www.mak.com/latency.pdf

[14] D. Nemeth, "Benchmarking the RTI for Use in a Simulated Radio". Spring 1999 Simulation Interoperability Workshop (SIW).

[15] Department of Defence, Defence Modeling and Simulation Office, "BmThroughput benchmark source code" Was available from http://www.dmso.mil

[16] P.Knight, A.Corder, R.Liedel, "Independent Throughput and Latency Benchmarking for the Evaluation of RTI Implementations". Fall 2001 Simulation Interoperability Workshop (SIW).

[17] P.Knight, R.Liedel, M.Klinner, "WBT RTI Independent Benchmark Tests: Design, Implementation, and Updated Results", Spring 2002 Simulation Interoperability Workshop (SIW).

[18] Department of Defence, Defence Modeling and Simulation Office, "BmTimeAdv benchmark source code" Was available from http://www.dmso.mil

[19] R. Fujimoto and P. Hoare, "HLA RTI Performance in High Speed LAN Environment". Fall 1998 Simulation Interoperability Workshop (SIW).

[20] J.Hung, M.Torpey, W.Civinskas, F.Hodum, "Performance Cost of Using Time Management Services". Spring 2002 Simulation Interoperability Workshop (SIW).

[21] A.C.Heurch, A.Horskotte, H.Rzehak, "Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux – Technical report", Department of Computer Science, University of Federal Armed Forces, Munich. Last obtained Nov 22, 2005 from: http://inf3-www.informatik.unibw-muenchen.de/research/linux/milan/paper_milan.pdf

[22] G.Herrin, "Linux IP Networking". Last obtained Nov 22, 2005 from: http://www.cs.unh.edu/cnrg/gherrin/linux-net.html

[23] A. Shankar , "Linux Networking" - Sep 2004. Last obtained March 25, 2006 from: http://limnos.csrd.uiuc.edu/notes/linux-networking/core.html

[24] O'Reilly Network, "Free BSD Basics – IP Protocol Layers Explained" March 2001. Last obtained Feb 3, 2006 from: http://www.onlamp.com/pub/a/bsd/2001/03/14/FreeBSD_Basics.html?page=1

[25] M.Lorenzo, M.Muuss, M.Caruso, B.Riggs, "RTI Latency Testing over the Defense Research and Engineering Network". Spring 2001 Simulation Interoperability Workshop (SIW).

[26] Perception without awareness: perspective from cognitive psychology. Cognition 79, 115-134. P. M. Merikle, D. Smilek, J. D. Eastwood, (2001).

Appendix A: Bibliography

1. J. O. Calvin, C. J. Chiang, S. M. McGarry, S. J. Rak, D. J. Van Hook, M. Salisbury, "Design, Implementation, and Performance of the STOW RTI Prototype (RTI-s)". Spring 1997 Simulation Interoperability Workshop (SIW).
2. Stephen R. Kolek, Bundling Presentation for the CAS Working Group at the Fourteenth Workshop on Standards for the Interoperability of Distributed Simulations, September 16-20, 1996. .
3. B. Dillman, J. Murphey, D. Bleichman, "Effects of high latency wide area networks in distributed simulation". Submitted for Spring 2006 Simulation Interoperability Workshop (SIW).

4. R. D. Wuerfel, F. J. Hodum, "RTI-NG Process Model". Spring 2001 Simulation Interoperability Workshop (SIW).
5. Introduction to the High Level Architecture. Last obtained Nov 22, 2005 from: http://www.ecst.csuchico.edu/~hla/LectureNotes/HLA_1.3NG_M1_P1.pdf

Appendix B:

This section contains a more detailed description of various concepts introduced throughout the research.

B.1 Access Latency of *getTimeOfDay()* in Linux OS

The access latency of the *getTimeOfDay()* system call is critical to the accuracy of time measurements, such as between sending and receiving a message via the loop-back federate. In Linux, a user thread has almost direct access to hardware resources. A call to *getTimeOfDay()* immediately generates a soft interrupt that transfers control to the Linux kernel code that can read the hardware clock. There is still some latency due to the need to switch from the context of the user thread to the interrupt context while saving thread variables and registers and then restoring them later on. Latency was measured using the following pseudo code:

Record start time;
Call getTimeOfDay()100000 times;
Record end time;
Access latency = (end time – start time) / 100000;

The access time for the experimental system was 0.0003 milliseconds.

B.2 Calculation of Network Hardware Utilization

d – data size (payload) per message in bytes

ipOH–IP layer overhead (40 bytes for UDP, 52 bytes for TCP)

rtiOH–RTI message overhead (176 for DMSO RTI attribute update, 24 for FDK attribute update, 26 for MAK RTI attribute update, 92 for DMSO RTI interaction, 12 for FDK interaction, 14 for MAK RTI interaction)

BR – Network layer bit rate (100*10^6 for 100BaseT Ethernet)

N – Number of messages per second

NpP – Number of bundled messages per packet

The utilization without data-bundling is calculated as follows

U = [N * (d + ipOH + rtiOH) * 8] / BR

With data-bundling there is just one ipOH for NpP messages and utilization is:

U = [N * (d + rtiOH) * 8 + (N / NpP) * ipOH. * 8] / BR

Appendix C: Abbreviations

ack	Acknowledge
AHVPS	Attribute Handle Value Pair Set
DND	Department of National Defence
DMA	Direct Memory Access
DMSO	Defence Modeling & Simulation Office
GPS	Global Positioning System
HLA	High-Level Architecture
IP	Internet Protocol
irq	Interrupt ReQuest
LAN	Local Area Network

LBTS	Lower-Bound Time Stamp
LRC	Local RTI Component
MTU	Maximum Transmit Unit
NIC	Network Interface Card
NTP	Network Time Protocol
PC	Personal Computer
RT	Real Time
RTI	Run-Time Infrastructure
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
TSO	Time-Stamp Ordered
WAN	Wide-Area Network