Real-time Scheduling of HLA Simulator Components

Roger Jansen, Wim Huiskamp, Jan-Jelle Boomgaardt, Marco Brassé

Command & Control and Simulation Division,
TNO Physics and Electronics Laboratory.
P.O. Box 96864
2509 JG The Hague, The Netherlands
E-mail: {jansen | huiskamp}@fel.tno.nl

Keywords:
Real-time Scheduling, RTI Performance, HLA, Simulator Components, Middleware

ABSTRACT: *For several years TNO has done research on the use of component based HLA federates. An individual simulator, participating in an HLA federation, can itself be composed of various components interacting through the same HLA interface as the overall federation. Simulator components communicate via a middleware layer called the Run-time Communication Infrastructure (RCI). The RCI is based on the HLA Run-Time Interface (RTI), but allows the use of other standards as well. Currently, HLA developers that are interested in real-time performance disregard most of the RTI functionality with respect to time management in an attempt to minimize the overhead of the RTI. The limited performance of the RTIs holds back potential HLA developers from building HLA compliant federates and it certainly raises doubts over the feasibility of HLA federations with real-time requirements, including component based federates. In the recent past, TNO has performed tests regarding performance of component based architectures that use the RCI in combination with the DMSO RTI. These results indicated the areas on which further research should focus, that is real-time HLA simulations and high-performance RTIs. In addition to the time management services of the RTI, the RCI middleware has been extended with a time triggered scheduling mechanism based on a globally synchronized wallclock time. This easy to use real-time scheduler can be of great support in the development of man-in-the-loop simulators. TNO has also developed a high performance RTI, which can easily use different communication media due to its layered design. The RCI middleware, RTI, and network has to provide a guaranteed 'Quality of Service' (QoS) to meet the real-time requirements of HLA federations. Therefore, TNO is in the process of extending the RCI with QoS policies. This paper discusses the results of our research into real-time scheduling and presents test results on a real-time Linux variant called RedHawk that prove the feasibility of a real-time simulator built as a Component Based Federate.*

1. Introduction

System cost is more and more defined by software development cost and less by hardware cost. Consequently, stronger reuse of existing (software) components is an important possibility to reduce development cost. Typically, reuse of components is easier when granularity of the parts is smaller. However, lower granularity also increases problems with respect to designing and implementing the correct (real-time) scheduling and communication between the different elements that make up the complete system. This paper discusses the problems and possible solutions for developing and implementing distributed systems based on cooperating components.

1.1 Objective

The TNO Physics and Electronics Laboratory (TNO-FEL) intends to use HLA in the area of 'real-time' federations and Component Based federates. Firstly, we want to prove that the HLA standard itself is a not the reason of the sometimes mediocre real-time performance of HLA federates. Since the RTI is not in the 'public domain', an in-house implementation had to be developed. This 'high-performance' RTI should meet the following requirements:

1. Comply with RTI 1.3 Interface Specification [1].
2. Feature a layered design, that allows easy exchange of data transport media.
3. Provide support for Real-Time functionality (deterministic behaviour).
4. Provide high performance in terms of bandwidth and latency.

The first prototype of our high-performance RTI was based on a local Shared Memory data transportation mechanism. Subsequently a real-time Distributed Memory RTI was implemented and tested. This network version of our RTI is based on Ethernet and runs under IRIX, Linux, SUN, and Windows. In this paper, we focus on a real-time version of Linux called iHawk that runs on low cost platforms. The results are compared to IRIX.

Using our RTI and middleware it will be shown that HLA compliant federates can be built to meet certain real-time constraints, such as an application to application latency of less than 1 milliseconds. We even want to go a step further and want to prove that it is possible to build a real-time simulator out of components that communicate through HLA as long as the middleware and its supporting RTI and communication network provide a guaranteed 'Quality of Service' (QOS).

1.2 Organization

The remainder of the paper is organized as follows. Section 2 discusses the Component Based Architecture and HLA middleware layer. Sections 3 and 4 discuss the real-time scheduling requirements and concepts used by this architecture. In section 5 our high performance RTI is described. Section 6 describes the communication media used by our RTI and section 7 describes the real-time Linux OS. In sections 8 and 9 several benchmark tests are described and the results are presented. Finally, section 10 contains our conclusions.

2. Component Based Architecture and HLA Middleware

Decomposing a simulator into a well-defined set of interacting components increases the potential for reuse and offers a natural work breakdown structure during federate development. Components are considered the basic building blocks of our simulators and can potentially be used in more than one type of simulator. Examples of (flight) simulator components are: a component that handles the Human-Machine-Interface (HMI), a flight dynamics component, a component that visualises the Out-The-Window virtual environment, or a motion platform component. Consequently, a simulator can be thought of as a set of interacting components. The total functionality of the simulator may be expressed as the 'sum' of the functionality of its constituting components.

The use of simulator components was formalised by TNO-FEL in the Component Architecture developed within the SIMULTAAN project [2]. A *Federation* is similar to an HLA Federation in every way. A *Federate*, however, now consists of a set of Components that may be distributed over several networked computer systems (See Figure 1). In fact, a Federate is regarded as a *federation of Components*. Similar to HLA, a Component is described by an HLA Object Model Template (HLA-OMT). This way, we capture and document all relevant interface agreements between Simulator Components within a component based federate in a similar way as the interface agreements between federates.

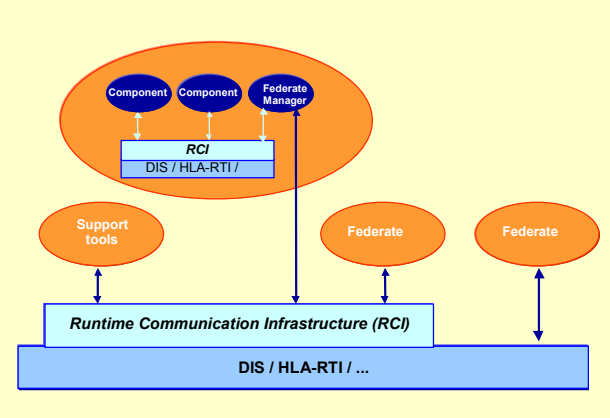

Figure 1 Component Architecture (schematic): each set of Components is managed by a Federate Manager, forming a Federate within the Federation. The RCI is TNO-FEL's middleware layer for interoperability.

All Components interact with the simulation environment through a standard interface provided by the Run-time Communication Infrastructure (RCI) Middleware Layer. The RCI provides the component developer with the necessary functionality to incorporate the Component into a Federate. In fact, the RCI interface abstracts from the federate and component level, so there is no limit on the number of decomposition levels.

The Federate Manager Component (FMC) is a special element of the Component Based Architecture whose role is to represent the set of Components to the overall federation as if it were a single federate, whose internal component structure is of no interest to the other federates. The FMC forwards all relevant data from the Components to other federates and vice versa. The FMC is able to do SOM to FOM mappings and performs coordinated state transitions, e.g. initialise, start simulation, and stop simulation. A code generator is

available to build an FMC which depends on the SOM, the FOM, and the State Transition Diagram.

The RCI has a protocol-independent interface to the simulated environment. It shields the federate developer from many technical details concerning the underlying communication standard and protocols, so one can focus on the actual simulation model rather than the lower level data exchange issues. Also it facilitates software porting of components to other communication standards, because the component uses the same API for different communication standards.

The RCI consists of two separate software layers, one is called the *Environment*, that reflects the current state of the Federate, and the other is called the *Communication Server*, that translates the events to a specific interoperability standard, e.g. DIS or HLA. (see Figure 2).

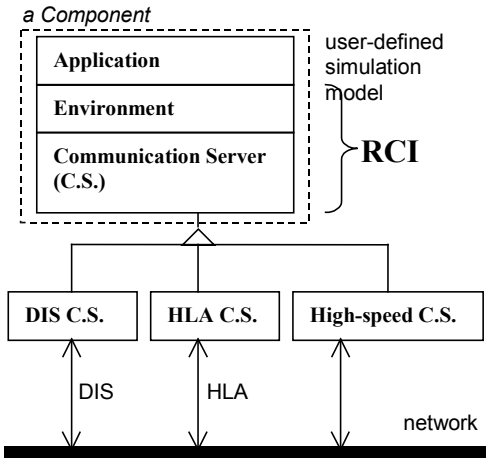

Figure 2 RCI Middleware structure

3. Real-time Scheduling

The real-time performance of the currently available RTI implementations (e.g. DMSO's RTI-NG) is not very impressive [5,8,9]. Especially when federates are hosted on different computers, the Application-to-Application latencies are often too large. Part of that latency is due to the network, but a significant part is suspected to be due to the RTI. Currently, HLA developers that are interested in real-time performance disregard most of the RTI functionality with respect to time-management in an attempt to minimize the overhead of the RTI. These doubts about the performance of the RTI holds back potential HLA developers from building HLA compliant federates and certainly raises doubts over the feasibility of HLA federations with real-time requirements, including simulators built as Component Based Federates as discussed in Section 3. TNO and its partners in the SIMULTAAN project performed earlier evaluations and tests regarding performance of the RCI in combination with the DMSO RTI with regard to component based architectures [7]. These results indicated the areas on which further research should focus. The remaining chapters of this paper discuss the results of that research.

3.1 What does Real-time Scheduling mean?

The objective of scheduling is to develop an adequate order of execution of tasks that is feasible. The schedule should meet the timing and resource requirements of the system.

Real-time means the ability to meet a deadline (e.g. response time to an external event). For a real-time system the response time to an event is just as important as the logical correctness of that response Real-time systems are generally split into two types depending on how serious their deadlines are and the consequences of missing one. These are :

- Soft real-time systems.
- Hard-real-time systems.

Soft real-time means missing an occasional deadline is acceptable. For example, a telephone switch might be permitted to lose or misroute one call in 100K calls under overload conditions and still be within specification. In contrast, even a single missed deadline in a hard real-time system is unacceptable, as this might lead to loss of life or an environmental catastrophe.

Specifying that a system is 'real-time' says nothing about processor performance, network bandwidth or system cost. A $1 microprocessor running at 4MHz may provide RT capabilities for a washing machine controller, while a $30K workstation running at 2.4 GHz could fail the RT demands of a traffic control system.

3.2 Real-time Requirements for a Component Based Simulator

A simulation must anticipate for a man-in-the-loop. This feedback loop constrains the time-period of the sum of the tasks required for each loop. The worst-case performance of the sum of the tasks must lie below the human interaction response.

Jitter is a term for timing deviation of cyclic events. It defines the difference between the maximum and the minimum timing of an action. Jitter is a measure for the quality of the timing. The human perception of a visual moving image requires an update of about 25-30Hz.

Various visual-views show the same simulations at the "same" time. For example a plan-view, a radar-display and a 3d-view. Our eye perceives the various streams as synchronous in time if a requirement of 30 Hz is met. The human eye is rather sensitive for jitter in this frame update rate.

The *TNO Fighter Federate (TFF)* developed by TNO-FEL and demonstrated at ITEC 2000 shows the Component approach for a generic fighter simulator [3]. The *TFF* consists of a number of Components using the RCI as interoperability layer. Some Components are based on Commercial Off-The-Shelf (COTS) software linked to the RCI (e.g. FLSIM), while others are in-house TNO developments (e.g. Visual). Typical communication requirements in this case are:

Cycle Times:
Control input: 100 Hz
Dynamic model: 100 Hz
Visual: 30 Hz update rate, 60 Hz refresh rate

Jitter:
1 ms scheduling accuracy

Latency (worst-case):
End-to-End Latency between components: 30 ms
End-to-End Latency between federates: 100 ms

Bandwidth:
3 Mbit/sec for a component based simulator with 2 cockpit stations, including displays, controls and visual and about 150 Computer Generated Forces.

We propose that two conditions must be met when (hard) real-time performance and deadlines are to be achieved in a distributed environment:

- The interoperability layer (middleware, RTI and network) must provide a defined QOS, which includes a bounded latency. Obviously, the OS and hardware platforms must also support these RT performance requirements.
- The federation must provide a system wide synchronized clock which can be observed with a known maximum error by any federate. The synchronized clock is the means by which the schedules of the different federates are kept aligned.

4. RCI Time Management Schemes

Time Management (TM) is concerned with the mechanisms for controlling the advancement of each joined federate along the federation time axis. In general, time advances shall be coordinated with Object Management (OM) services so that information is delivered to joined federates in a causally correct and ordered fashion.

All time references in the TM and OM RTI service calls (type RTI::FedTime) are pure logical times and have no relation to any kind of *wallclock time*. The simulation time and timestamps in the RCI are also logical times, but the RCI provides also the concept of wallclock time. Note that in this context the term *real-time* is often used instead of wallclock time, but we relate real-time only to the capability of meeting deadlines. The RCI provides a service to set on and off pacing the simulation time to the wallclock time. When pacing is off, the simulation is executed as fast as possible; when pacing is on the simulation runs *n times real-time*, where *n* is the provided timescale, which defaults to 1. The RCI will derive the 'wallclock' time from the local system-time, which may in turn be synchronized to a globally synchronized time.

The RCI supports different kinds of time management schemes. Each scheme is represented by an RCI scheduler, which a federate developer can easily select for his application. The RCI Time Management schemes are classified as follows:

1. *No Time Synchronisation*, in which each federate advances time at its own pace i.e. the simulation time is not synchronized among the federates. Updates shall be sent without a timestamp. The scheduler that supports this scheme is called *Default Scheduler*.
2. *Conservative synchronisation*, in which federates send time stamped events (never less than current time + lookahead) and advance time only when it can be guaranteed they will receive no past events. The simulation time shall be synchronized among all federates. Two types of schedulers support this scheme:

    - *HLA Time Management (HLA-TM) Scheduler.* Conservative scheduling is guaranteed by the RTI.
    - *Universal Time Coordinated Time Management (UTC-TM) Scheduler.* Conservative scheduling is guaranteed by pacing the simulation time to a globally synchronized time.

HLA-TM Scheduler
This scheduler uses the HLA-TM services of the RTI, like "Enable Time Regulation", "Enable Time Constrained", "Time Advance request", and "Next Event Request". The time advancements of all federates are coordinated by the RTI and they have no relation to any kind of wallclock time. A time advancing federate has to wait until the RTI delivers a "Time Advance Grant".

Therefore, the federate developer has no guarantee the simulation time will be updated before a certain deadline

An ideal place to do the time coordination is the central Fedex process, which is started up by the RTIexec process for each created federation. Regardless of how HLA-TM is implemented, it is obvious that an HLA-TM enabled simulation will have more RTI overhead than one without using HLA-TM. Besides this overhead, a more fundamental disadvantage of using HLA-TM in real-time simulations is identified: meeting a time advancement deadline for a specific federate is not independent from other federates. If, for any reason, a federate is not able to request a time advancement in time, then it can slow down the whole federation.

UTC-TM Scheduler
When using this scheduler the wallclock equals the UTC time, which is the same for all federates in the federation. Since our RTI implementation has to comply with the RTI 1.3 Interface Specification, it was chosen to implement this concept in the RCI middleware layer and leave the RTI API as is. The RCI puts a timestamp in the tag field of the RTI services "Update Attribute Values" and "Send Interaction" in order to deliver all messages in time stamp order as the HLA TM scheduler does.

The main difference between both conservative schedulers is that the UTC-TM Scheduler does not wait on the "Time Advance Grant" of the RTI, but it uses a time triggered scheduling mechanism. This way, all deadlines can be met. Of course, the other side of the coin is that the UTC-TM Scheduler in itself cannot guarantee that all time stamped messages have been received before the simulation time will be updated. A federation wide schedule that meets all timing and resource requirements of the system has to be defined also.

The way the RCI deals with the concept of scheduling of actions is not scheduler specific, but it matches best with the qualities of the UTC-TM Scheduler. Since it is very easy to switch between schedulers, nothing stands in the way to examine the feasibility of a schedule using the HLA-TM Scheduler.

The RCI divides the time axis in equally sized simulation frames. Within a simulation frame, the application can schedule multiple *Update, Output* and *Sync Actions.* These actions shall be repeated each frame at the specified *scheduled times*. Only the Update Action has effect on the simulation time. Figure 3 shows the relationship between the scheduled time and the simulation time in one frame. The frame frequency can be set independently from the scheduled times.

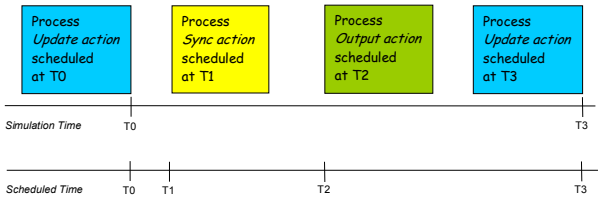

Figure 3 Action processing within one frame

Update Action
An Update Action reads remote data from the federation the federate is subscribed to, updates remote objects, processes application subscribed callbacks and advances to the requested simulation time, which is equal to the scheduled time (T0).

Output Action
An Output Action sends attribute changes of local objects into the federation at the scheduled time (T1).

Sync Action
A Sync Action triggers a special user defined callback. Using this mechanism the application gets processing time at the scheduled time (T2).

The scheduling of actions can be specified:
1. in the *schedule sheet* (ASCII file containing all parameters) or
2. as an argument to the Control service of the RCI. In this case the RCI shall spread the actions evenly across the frame.

5. High Performance RTI

The RTI architecture developed by TNO consists of three major elements :

- a (static or dynamic) RTI library, which has to be linked together with the application source code,
- the Fedex, which is a separate process that keeps track of federation-wide administration,
- the RTIexec, which launches a Fedex process for each created federation.

The TNO RTI has exactly the same API as the DMSO RTI NG, since the same C++ header files and library names are used. It has to be stressed that the current prototype of the RTI is only a partial RTI implementation, since less common RTI services like Ownership Management and Data Distribution Management are not implemented yet.

The RTI is designed in a layered way, such that the lower layers can easily be replaced when another

communication medium is chosen. Currently, two kinds of communication media are supported:

- Shared Memory using Message Queues (RTI-SM).
- Distributed Memory using UDP/IP Ethernet Sockets (RTI-DM).

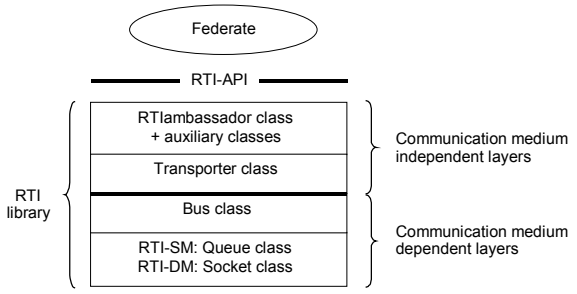

Figure 4 Layered TNO RTI design

Our layered RTI design and Component Based Architecture makes it easy to port a simulator to another system configuration. The RTI runs on several Operating systems, like IRIX, SUN, Linux, and Windows. In this paper, we want to focus on PC-like platforms and general purpose networks to prove the concept of a HLA Component Based Simulator. A real-time Linux variant called RedHawk Linux seems to meet all our real-time constraints as we will see later on in this paper. Ethernet is a natural candidate for our RTI-DM, because it has been accepted as proven technology. Due to the layered design of our RTI it is rather easy to use other industrial networks like CAN bus or Profibus if a general purpose network is not feasible.

6. RTI communication media

6.1 Ethernet

Ethernet was originally developed in the 1970s and the standard is now known as IEEE 802.3. Ethernet, especially the 100 Mbps twisted pair version is very popular and low-cost due to its widespread use in office and research environments. Ethernet uses Carrier Sense, Multiple Access, Collision Detect (CSMA/CD), which means that transmitting stations broadcast all messages. The receiving station responds to the message when it detects its own address as destination. . Stations may start transmission when the network is idle. Transmitting stations listen to their own messages and collisions are detected by comparing the transmitted data with the data 'on the wire'. Collisions will cause the transmitting stations to wait for a randomly determined delay time before trying again. Thus normal Ethernet solves the collision problems probabilistically and can not guarantee timely access.

Ethernet defines only the (OSI) 'physical layer' and 'data link layer' and not the protocols above these layers. Internet Protocol (IP) defines the 'Network layer' and runs on top of the datalink and physical layer. Transmission Control Protocol (TCP) and Universal Datagram Protocol (UDP) are in turn layers on top of IP (Transport and Application layers). The average PC will typically show delays in the order of 200 us introduced by processing the payload data according to the Ethernet protocol stack. The transmission time over the wire for a message of 1500 bytes is around 120 us (at 100 Mbit/s).

Note that modern Ethernet is often based on 'switches' that provide smart switching of IP-packets based on addresses and thus avoids collisions and allows all nodes to run at maximum bandwidth instead of sharing this bandwidth with all other nodes. However, in order to prevent collisions, a switch will use internal buffering and introduce additional and unpredictable latencies (~10 us). Switches may also introduce a change in the delivery order of messages.

6.2 Shared Memory

RTI-SM communication is based on message queues. The fedex and each federate have their own message queue for incoming messages. A sender is able to enqueue a message on the queues of all interested receivers. Messages are removed from each queue as soon as they are read.

For safe data transfer the receiver and each sender needs exclusive access to a shared memory location. This lock mechnism is implemented by a semaphore for each queue.

Using our message, queue approach data is duplicated for each receiver. However, in case of a memory block shared by all readers, extra overhead is needed to determine when all interested readers have read the message before dequeuing it. In [10] a hybrid approach is presented where the data is stored at one common place and where each reader has still its own queue in which references are stored to the real data. Actually, the best approach depends on the message sizes and the data flow of the federation. For the moment, there is no need to optimize our approach.

7. Real-time Linux

Several real-time Linux variants are on the market nowadays. RTLinux is well known and provides a seperate real-time kernel that controls real-time processes. Linux runs as a task under the real-time kernel. Disadvantages are the strict distinction between real-time and non real-time processes and a separate, non

standard Linux API for the real-time processes. This API is rather limited and introduces portability problems.

RedHawk Linux of Concurrent Computer Corporation is based on the popular RedHat Linux distribution. It provides determinism for the entire application environment, including file I/O, network, and graphics. RedHawk has a single, fully pre-emptible, kernel and provides a single API which is fully compatible with RedHat. It provides load-balancing and CPU shielding to maximize determinism and real-time performance [4]. The RedHawk Operating System runs on iHawk high-performance computers, each having one or more Intel Pentium Xeon processors.

RedHawk Linux supports a dedicated Real-time Clock & Interrupt Module (RCIM). This hardware module includes a high resolution synchronized clock and interrupts. RCIMs can be connected by a cable such that a reliable common time base is achieved. Such a system wide clock is one of the conditions for real-time scheduling. The RCI UTC-TM Scheduler could be based on this RCIM clock.

7.1 Clock Synchronization

Apart from using a separate clock synchronization cable (e.g. the RCIM approach), a more general solution to implement a distributed clock is the Network Time Protocol (NTP). In this case, time messages are used to synchronize the local clocks, which are sent on the same network used for other communication. Note that systems like FlexRay and some of the Time Triggered Protocols use the actual data messages themselves to re-synchronize the local clocks. For wide-area networks a synchronization mechanism based on GPS receivers is also a possibility. The GPS devices transmit a highly accurate UTC timestamp and also provide several sync signals (e.g. PPS - Pulse Per Second).

8. System Limits Measurements using RTIs on SGI/IRIX and iHawk/Redhawk Linux

The purpose of this benchmark test is to determine the maximum throughput on a specific system configuration for a pipeline of three HLA federates. This configuration is chosen because it corresponds to the critical part of a (flight) simulator.

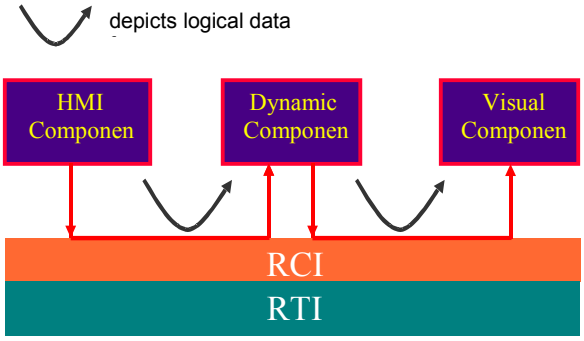

Figure 5 Pipeline of three Federates

The RCI of a sending federate puts the local system time in the special RTI tag field every time an object update is sent to the RTI. The tag is a string which is not interpreted by the RTI and may be used to communicate federation specified information about the update. This way, the FOM need not to be extended for latency logging purposes. The RCI of the receiving federate stores the timestamp retrieved from the tag and also its current local time in memory each time an object update is received. Special RCI service calls are used to set on and off this logging functionality and to write all measured end-to-end latencies to file after the test has been completed.

Several RTIs on several operating systems have been tested. Besides the DMSO RTI, the TNO RTI is tested using different communication media. The RCI/RTI Benchmark test configuration using shared memory and the one using a network look very similar:

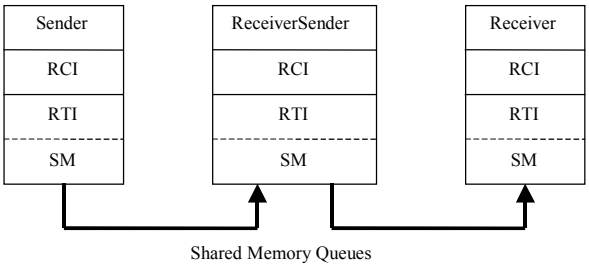

Figure 6 RTI-SM using Shared Memory Queues

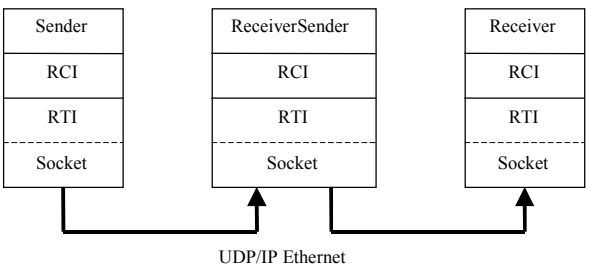

Figure 7 RTI-DM usingUDP/IP Ethernet

The Sender federate sends data with a fixed frequency to the ReceiverSender Federate, which sends the data as fast as possible to the Receiver federate. Both the ReceiverSender and the Receiver federate measure the latency of each received message, and thus also the mean and maximum latencies. They also measure the mean read frequency.

Several runs are needed to determine the system limits. The send frequency will be incremented after each run as long as the mean read frequency of the Receiver federate is at least the send frequency of the Sender federate and as long as at most 0.1% of the data is lost.

The data size of the message at application level is 512 bytes. This is without any RCI/RTI overhead like the tag string and the object and attribute handles. For the shared memory RTI the total packet size is 586 bytes; for the distributed RTI the total network packet size is 622 bytes, which includes the UDP/IP header of 28 bytes.

For the distributed tests, it is best to let the Sender and Receiver federate run on the same system, because then we know for sure that the end to end latency is measured accurately with the same system clock.

8.1 Test configuration 1a (DMSO RTI; IRIX).

System configuration: DMSO RTI 1.3 NG3.2 on two SGI IRIX 6.5 ONYX systems. Sender and Receiver federate run on the same system each at a shielded 250 Mhz CPU. The ReceiverSender federate runs on another system at a shielded 195 Mhz CPU. A general purpose network (EBF-LAN) is used, in which all machines are connected by a 100 Mbps Switch.

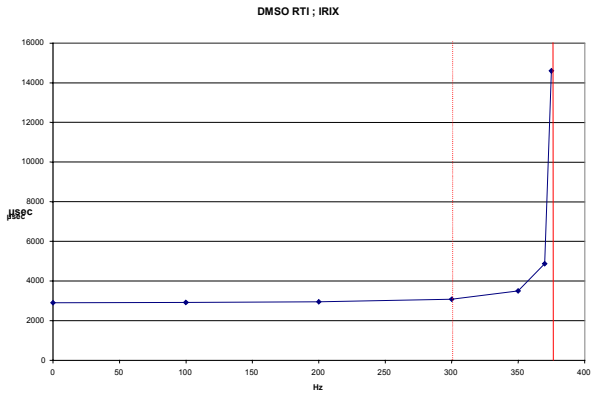

Figure 8 DMSO RTI on IRIX

F_{sender} < 300 Hz : average latency increases slightly from 2900 to 3080 µsec.
300 < F_{sender} < 375 Hz : average latency increases gradually to 14600 µsec.
F_{sender} > 375 Hz : unacceptable packet loss

8.2 Test configuration 1b (RTI-DM; IRIX).

System configuration: same as test 1a, but now the DMSO RTI is replaced by the RTI-DM version.

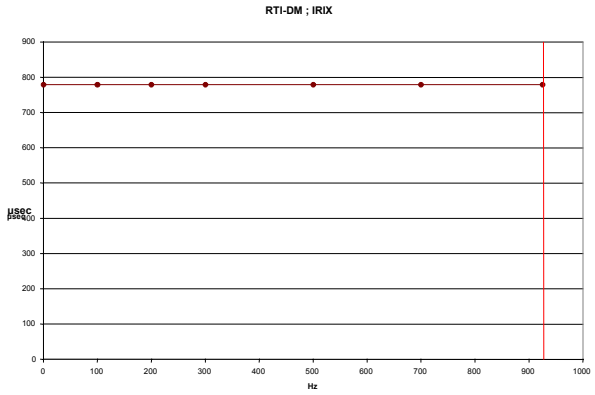

Figure 9 RTI-DM on IRIX

F_{sender} < 925 Hz : constant average latency of 780 µsec.
F_{sender} > 925 Hz : unacceptable packet loss

We see that on exactly the same system configuration our own RTI performs much better than the DMSO RTI. The average latency is lower and the maximum frequency is higher.

8.3 Test configuration 2 (RTI-SM; IRIX).

System configuration: RTI-SM on an SGI IRIX 6.5 ONYX system. Each federate runs at a shielded 250 Mhz CPU.

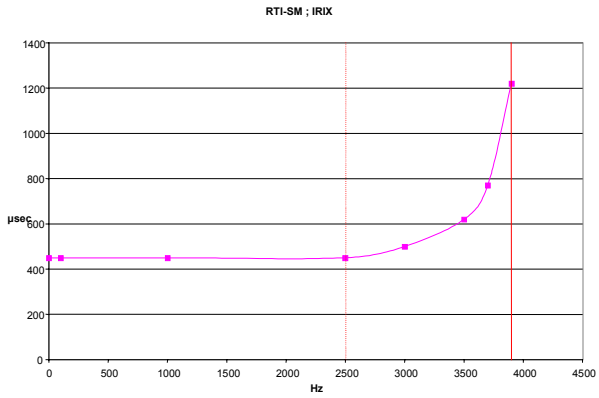

Figure 10 RTI-SM on IRIX

F_{sender} < 2500 Hz : constant average latency of 450 usec.
2500 < F_{sender} < 3900 Hz : average latency increases gradually to 1200 usec.

$F_{sender} > 3900$ Hz : $F_{receiversender} < F_{sender}$

We see that when the communication medium of our RTI changes from Ethernet to Shared Memory the average latency is again lower, while the maximum frequency is higher.

Further analysis of the log files shows that the ReceiverSender federate reads and sends messages in bursts of up to 7 packets when the frequency becomes higher than 2.5 kHz. The minimum latency is then still acceptable, but the high average latency is determined by the high latencies caused by the oldest packets in the bursts. The ReceiverSender federate is obviously the bottleneck, because it has to read and send packets, while the other two federates only have to do one job.

The SGI IRIX computers used in the previous test configurations are rather old, especially when their CPU rate is compared to current CPU rates at the PC market. The following tests are performed on more state of the art RT Linux PC systems.

8.4 Test configuration 3a (RTI-DM; iHawk; hub).

System configuration: RTI-DM on two RedHat 8.0 RedHawk Linux iHawk systems. Sender and Receiver federate run on the same system each at a shielded 700 Mhz CPU. The ReceiverSender federate runs on another system at a shielded 2400 Mhz CPU. The two machines are connected by a 100 Mbps Hub.

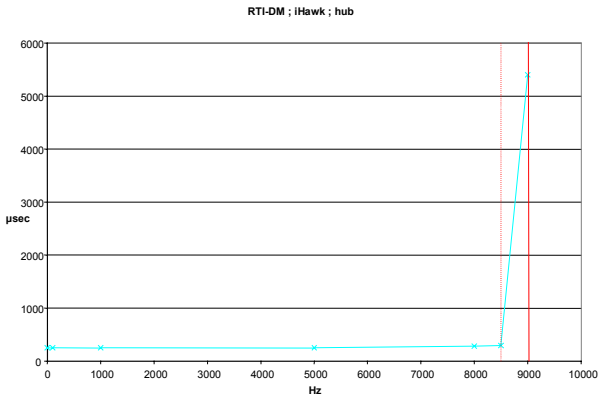

Figure 11 RTI-DM on iHawk using a hub

$F_{sender} < 8500$ Hz : more or less constant latency of 270 usec.
$8500 < F_{sender} < 9000$ Hz : average latency increases gradually to 5400 usec.
$F_{sender} > 9000$ Hz : unacceptable packet loss.

The maximum frequency of 9 kHz corresponds with the maximum bandwidt of 100 Mbit/s. The one way datarate is $9000 \cdot 622 \cdot 8 = 45$ Mbit/s. For both ways this is 90 Mbit/s, which means that the ethernet fills up and packets get lost by too many collisions.

8.5 Test configuration 3b (RTI-DM; iHawk; switch).

System configuration: same as test 3a, but now the two machines are connected by a 100 Mbps Switch.

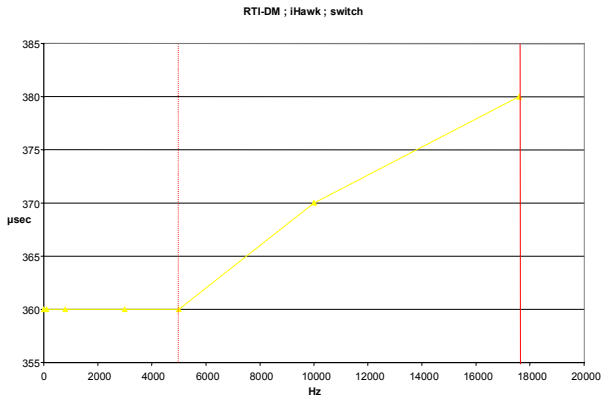

Figure 12 RTI-DM on iHawk using a switch

$F_{sender} < 5000$ Hz : constant average latency of 360 usec.
$5000 < F_{sender} < 17600$ Hz : average latency increases slightly to 380 usec.
17600 Hz is the maximum frequency of the Sender federate.

Since the CPU of the ReceiverSender federate runs on is much faster than the CPU the Sender federate runs on, the bottleneck of the pipeline is not the ReceiverSender federate. The maximum send is limited by the 100 Mbps network.

We see that that the latency, in case of the switch, is higher than with the hub in test 3a. This makes sense, because a hub is a simple repeater of data while a switch stores and forward the data. However, the maximum frequency is twice as high for the switch. The switch can isolate both data flows and improve the performance. Each link has in fact a bandwidth of 100 Mbit/s, which explains the maximum frequency of about 18 kHz.

8.6 Test configuration 4 (RTI-SM; iHawk).

System configuration: RTI-SM on a RedHat 8.0 RedHawk Linux iHawk system. Each federate runs at a shielded 700 Mhz CPU.

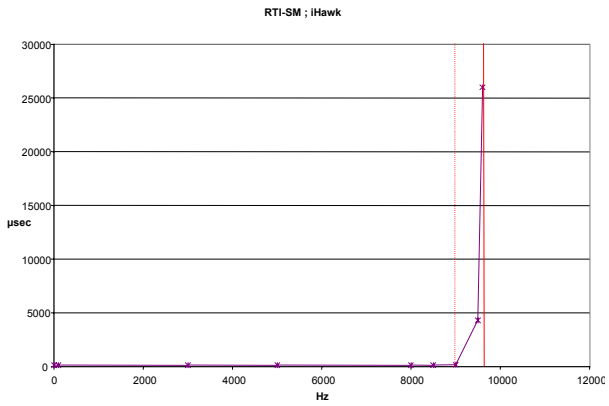

Figure 13 RTI-SM on iHawk

F_{sender} < 8500 Hz : constant average latency of 110 usec.
8500 < F_{sender} < 9600 Hz : average latency increases gradually to 26000 usec.
F_{sender} > 9600 Hz : $F_{receiversender}$ < F_{sender}

Compared to the distributed tests on iHawk (tests 3a and 3b), we see a lower latency when the send frequency is below 8.5 kHz. When the frequency is higher, the average latency becomes very high. The log files show that the buffer between the Sender and ReceiverSender federate fills up. The ReceiverSender federate cannot handle the incoming packets in time, because it runs on a much slower CPU than in tests 3a and 3b.

Compared to the the shared memory test on IRIX (test 2), we see a lower latency and a higher maximum frequency. This can be explained by the difference in CPU power. In both tests the ReceiverSender federate is the bottleneck when the frequency becomes high. The reading of packets in more or less regular bursts as we saw on IRIX is not observed now. On iHawk the buffer of the ReceiverSender federate fills up much more, which explains much higher mean latencies when the frequency is very high.

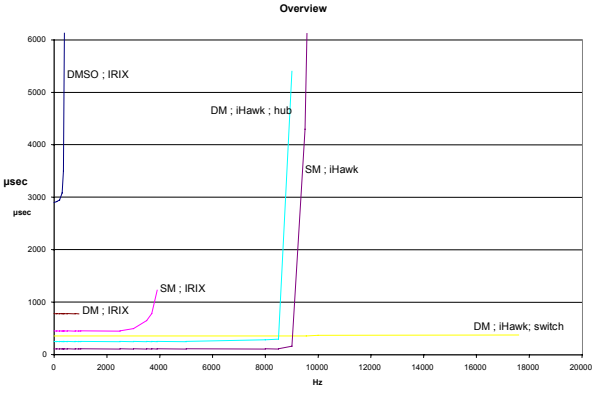

Figure 14 Overview

9. Jitter Measurements on SGI IRIX and iHawk RT Linux

Two kinds of jitter have been measured, i.e. jitter in local scheduling and jitter in network latency. For real-time distributed scheduling both have to be as low as possible. These tests have been performed under normal conditions unlike the stress tests of Section 6.

9.1 Local scheduling Jitter

The scheduling of a task on an IRIX 6.5 ONYX system at a shielded 250 Mhz CPU can be done accurate most of the time, i.e. the jitter is smaller than 10 µsec. However, although the process runs at a shielded CPU, the jitter over a rather long period of 10000 measurements was 39 µsec due to some sporadic peaks.

The scheduling of a task on a RedHat 8.0 iHawk system at a shielded 700 Mhz CPU after a sleep period followed by a shorter busy wait loop can be done very accurate. The jitter over a period of 10000 measurements was 3 µsec.

9.2 Network Latency Jitter

Network latency jitter has been measured by two simple process that both can send and receive UDP/IP packets. In fact only the lowest layer of the RTI-DM has been used. The data size of the message at application level is 16 bytes. The total network packet size is 44 bytes.

The network latencies between two IRIX systems connected by a 100 Mbps network is shown in Figure 15.

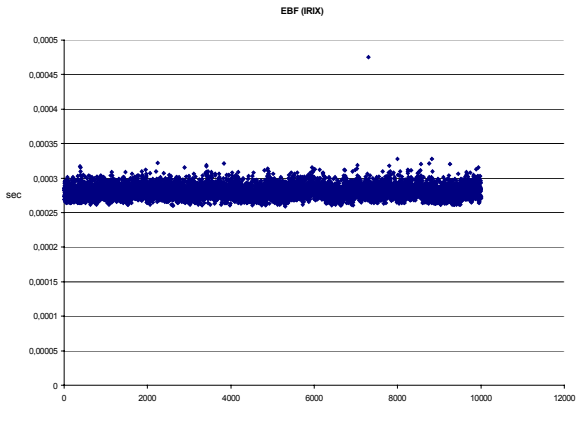

Figure 15 Network latencies on IRIX

The mean latency is 281 µsec. The minimum and maximum latency measured are 259 and 475 µsec, respectively. This means the jitter in the network latency is 216 µsec.

The network latencies between two iHawk systems connected by a 100 Mbps network is shown in Figure 16.

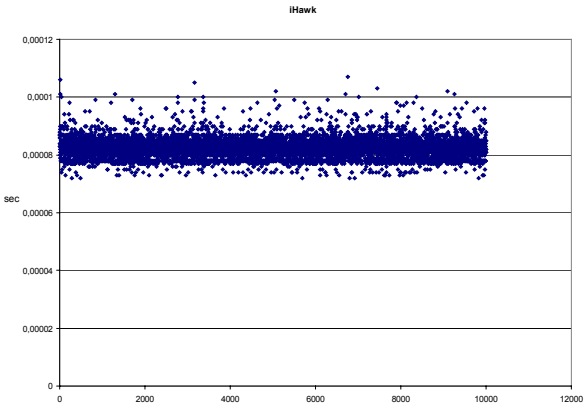

Figure 16 Network Latencies on iHawk

The mean latency is 82 µsec. The minimum and maximum latency measured are 72 and 107 µsec, respectively. This means the jitter in the network latency is 35 µsec.

10. Conclusion and Future Work

This paper has focussed on real-time requirements of our RCI middleware and on benchmark performances of several platforms, including the network, the hardware platform, the operating system, the drivers and the RCI/RTI. Table 1 gives some typical latency indications from application-to-application on distributed nodes. The hardware can not be compared as such, because of differences in clock-frequencies. However, given the notion that the iHawk platform is about three times faster than the SGI platform, this does not account for all the latency improvement. The iHawk/RedHawk performance indicators also definitely benefit from implementation aspects that you might expect from a real-time platform, like processor shielding, efficient task scheduling and efficient kernel use. One of our plans is to perform the same benchmark tests using the the MÄK RTI [6] and compare the test results with the results presented in this paper.

TNO-FEL will continue its research and development of RT component based architectures and define interface requirements for a supporting middleware. An important issue is the guaranteed QoS to be provided by the interoperability layer. HLA supports only a very small set of QoS policies, like reliable vs. best effort communication. We want to provide the federate developer with QoS policies to guarantee cycle times, latency, jitter and bandwidth. OMG Data-Distribution Service (DDS) [11] is just like HLA a publish-subscribe commmunication architecture, but targets more on real-time systems and specifies QoS semantics lacking in HLA. On the other hand, DDS has no simulation specific services like (logical) time management and federation synchronization, but the RCI middleware layer could implement these on top of DDS.

	Performance Indicators		
	Latency (µs)	Local Jitter (µs)	Network Jitter (µs)
SGI/IRIX 250 MHz		39	216
DMSO RTI	3000		
RTI-DM	780		
RTI-SM	450		
iHawk/RedHawk 700 MHz		3	35
RTI-DM	270-360		
RTI-SM	110		

Table 1 Typical Performance Indicators

11. References

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

[2] Marco Brasse, Wim Huiskamp, Olaf Stroosma, “A Component Architecture for Federate Development”, Simulation Interoperability Workshop, Fall 1999

[3] Wim Huiskamp, Henk Janssen, Hans Jense, "An HLA based Flight Simulation Architecture", Published in Proc. AIAA Modeling and Simulation Technologies Conference, Denver, Colorado, USA, August 2000. (AIAA-2000-4401)

[4] Steve Brosky, Steve Rotolo, “Shielded Processors: Guaranteeing Sub-milisecond Response in Standard Linux”

[5] Maximo Lorenzo, Mike Muuss, Mike Caruso, Bill Riggs, "RTI Latency Testing over the Defense Research and Engineering Network", 01S-SIW-081
[6] Douglas D. Wood, Len Granowetter, "Rationale and Design of the MÄK Real-Time RTI", 01S-SIW-104
[7] Arjan Lemmers, Paul Kuiper, Rene Verhage, "Performance of a Component Based Simulator Architecture using the HLA Paradigm", AIAA-2002-4476.
[8] Herbert Tietje, "Benchmarking RTIs for Real-Time Applications", 03E-SIW-026
[9] Brad Fitzgibbons, Thom McLean, Richard Fujimoto, "RTI Benchmark Studies", 02S-SIW-105
[10] Paul J. Christensen, Daniel J. Van Hook, Harry M. Wolfson, "HLA RTI Shared Memory Communication", 99S-SIW-089
[11] Rajive Joshi, Gerardo-Pardo Castellote, "A Comparison and Mapping of Data Distribution Service and High-Level Architecture"

Author Biographies

ROGER JANSEN is a member of the scientific staff in the Command & Control and Simulation Division at TNO-FEL. He holds an M.Sc. degree in Computing Science and a Master of Technological Design (MTD) degree in Software Technology, both from Eindhoven University of Technology, The Netherlands. He works in the field of distributed simulation and his research interests include distributed computing and simulation interoperability.

WIM HUISKAMP is a research scientist in the Command & Control and Simulation Division at TNO-FEL. He holds a M.Sc. degree in Electrical Engineering from Twente University of Technology, The Netherlands. He works in the field of High Performance Computing and Networking and specialises in design and implementation of distributed computer architectures. His research interests include system architecture, real-time simulation and autonomous vehicle technology. Wim was the project lead for several national and international simulation (and simulation interoperability) projects.

JAN JELLE BOOMGAARDT is a research scientist in the Command & Control and Simulation Division at TNO-FEL. He holds an M.Sc. degree in Software Engineering from Twente University of Technology, The Netherlands. He works in the field of High Performance Computing and Networking and specialises in design and implementation of distributed computer architectures.

MARCO BRASSE is a former member of the scientific staff in the Simulators Group at TNO-FEL. He is involved in several national and European projects focusing on HLA development. He holds an M.Sc. degree in Computing Science and a Master of Technological Design (MTD) degree in Software Technology, both from Eindhoven University of Technology, The Netherlands. His research interests include parallel and distributed computing, component based architectures, and embedded systems.