DSC North America 2003 Proceedings, Dearborn, Michigan, October 8-10, 2003 (ISSN 1546-5071).

PERFORMANCE EVALUATION OF A FRAMEWORK FOR DISTRIBUTED REAL-TIME DRIVING SIMULATION APPLICATIONS USING WINDOWS® BASED PCS

Yiannis Papelis, Ph.D.
The University of Iowa, NADS
2401 Oakdale Blvd.
Iowa City, IA 52242
(319) 335-4597
(319) 335-4658 fax
yiannis@nads-sc.uiowa.edu

ABSTRACT

The ever-decreasing cost of personal computers (PCs) has made them attractive platforms on top of which to design and implement driving simulators. Combined with the rapid advancement of processor power and the availability of powerful graphics cards, driving simulators built exclusively on commodity PCs are feasible and becoming commonplace. Furthermore, networking using affordable Ethernet hardware makes it possible to combine the power of multiple PCs and provide systems with aggregate capabilities that were unheard of only a few years ago. Related to the applicability and performance of these systems is the choice of operating system. The choices include dedicated real-time operating systems, Linux, and Windows ® NT/2000/XP, all of which have their respective advantages and disadvantages. Of primary concern is the degree of determinism afforded by each of these choices, as well as the availability and cost of development tools. This paper provides an architectural framework that supports distributed real-time systems, with special emphasis on driving simulation applications built on top of commodity PCs with focus on the applicability of PC hardware Windows NT ® and later. Detailed performance figures are provided for determinism of the system, both in stand-alone as well as distributed modes.

REAL TIME FRAMEWORK HISTORICAL OVERVIEW

The real time simulation framework described here is largely derived from the ICON [1] software framework utilized at the Iowa Driving Simulator (IDS) [2] and the enhanced version of it that is currently in use at the NADS [3]. A comprehensive review of the IDS and NADS real-time frameworks is beyond the scope of this paper; however, a short summary is provided.

Under this framework, a simulator is comp osed of multiple real-time subsystems all running under a fixed periodic schedule. The actual decomposition of the system into subsystems is not explicitly specified, and in fact varies greatly between different implementations. However, it typically involves grouping of functions that minimize data communication. Example subsystems include vehicle dynamics, scenario control, instrumentation, audio, etc.

Note that the framework does not specify which computer is running a given subsystem. In general, the framework is responsible for hiding such details from the subsystems. The IDS utilized an eight-processor Harris Nighthawk ® host (now manufactured by Concurrent Computer) along with various additional real-time computers that included an Alliant FX/2800 system, an HP quad processor system, and a Silicon Graphics Onyx Subsystems were mapped on individual processors on the main host or could be residing on any of the remote hosts.

Figure 1 illustrates a high-level block diagram that is applicable to both the IDS and NADS real-time fra mework. The key features illustrated are the decoupling of the subsystems from the details of the real-time scheduling implementation and the use of communication cells that act as blackboard type communication, in effect decoupling subsystems from each other.

The use of the communication cells in particular has been very successful in providing great configuration flexibility because it avoids point-to-point dependencies among subsystems. Under this model, individual subsystems do not communicate directly with each other; instead, they read their inputs from the input cells and write the output to the output cells. This makes it possible to change the way data is produced with very little ripple effect on the remainder of the system. For example, consider a subsystem such as audio that requires the velocity of ownship as well as the engine RPM. Typically, such output variables are produced by the vehicle dynamics subsystem. Now consider the situation where the dynamics model subsystem is broken into two subsystems, one simulating the chassis and one

simulating the engine. If the audio reads its data by any means that references the dynamics subsystem, then splitting the dynamics would require changing the audio subsystem as well. Under the framework described here, such changes do not propagate since both pieces of data exist in the communication cells and the audio only references the data, not the producing subsystem.

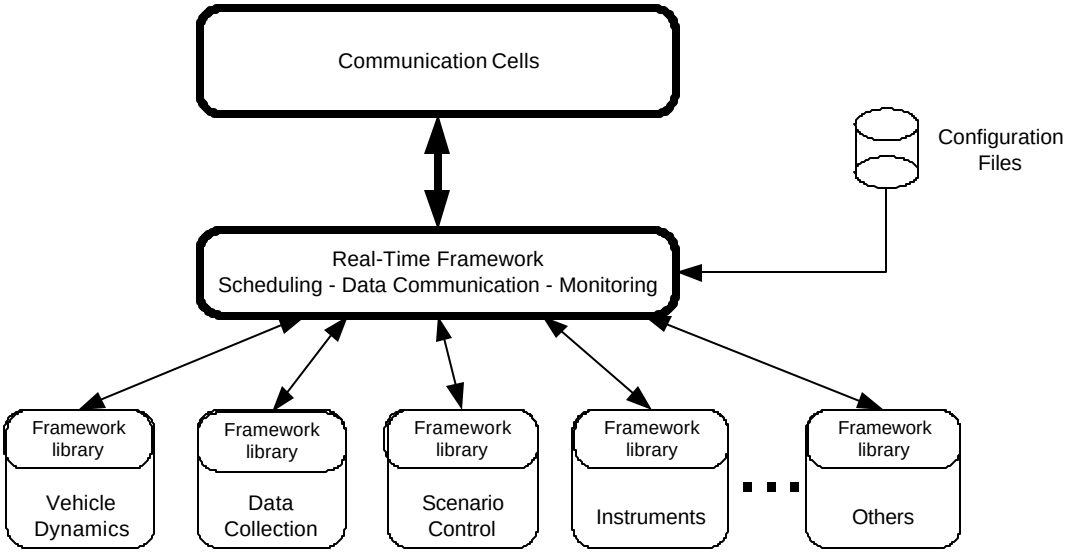

FIGURE 1 Legacy real-time framework block diagram.

To implement the real-time scheduling necessary for the simulator, the framework utilizes a fixed-priority, static, periodic schedule. One computer is responsible for providing the scheduling tick signal, which is intercepted by the framework library and awakens each subsystem at its allocated time slot. During each execution period, input cells are read automatically before subsystem execution begins and output cells are flushed automatically when execution ends. Each subsystem consists of a short initialization function followed by an endless loop that blocks on a function provided by the framework. Once the scheduling time has arrived, the blocking function returns the current state, allowing the subsystem to perform state specific computation. Once the computation is done, the blocking function is called again, repeating the cycle forever. Figure 2 illustrates one method for visualizing the relative execution timing of the various subsystems, as specified by the schedule. Each subsystem is assigned a start frame and period, and the overall schedule is specified using the major and minor periods. The scheduling tick signal provides the minor tick invocation. In the example shown in Figure 2, subsystem A would have a start frame of 0 and a period of 3, and subsystem C would have a start frame of 3 and a period of 6. If the timing tick signal was configured for a 5 mSec interval, that would provide an execution frequency of 66.66Hz for subsystem A, 200Hz for subsystem B, and 33.33 Hz for subsystem C. Note that the execution period of any one subsystem cannot be larger than the number of minor periods within a major period.

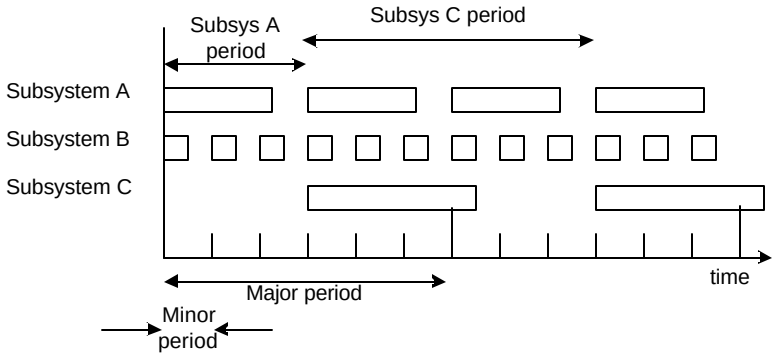

FIGURE 2 Visualization of static period schedule.

Figure 3 illustrates the communication pattern for subsystems A and C. Data is read once a subsystem is awakened and written right before the subsystem terminates its execution for the current frame. Note that it is up to the system

designer to put together a schedule that maintains the proper inter-subsystem data flow; however, individual subsystems do not have to worry about such details.

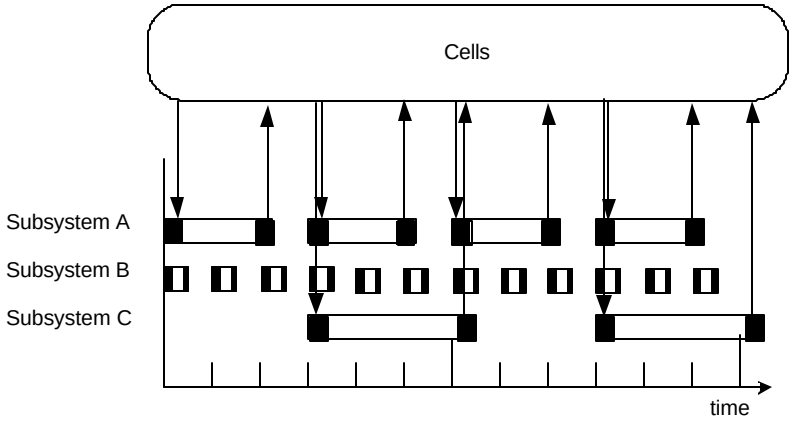

FIGURE 3 Subsystem communication pattern.

In its early phases, communication cell implementation on the IDS depended heavily on the shared memory multiprocessor architecture of the Nighthawk augmented by other host-to-host interfaces (DR11W, reflective memory) for passing data across hosts. The NADS utilizes Scramnet® reflective memory for both scheduling and data communication.

As initially delivered, NADS utilized a shared-memory multiprocessor Concurrent Computer with 8 CPUs as the main host. One unfortunate design decision made by the NADS implementers was utilizing two distinct sets of communication cells. One was mapped on the Concurrent shared memory and one was mapped on the Scramnet® board. The VME bus was not fast enough to allow all the data communicated between subsystems on the main host to also flow across the Scramnet®, thus the decision to have two distinct cells areas. While this decision provided a short-term solution to the problem, it violated the basic premise of the framework in that it created explicit dependencies among subsystems. Moving a subsystem to a different host could break a simulation unless code was changed so the subsystem read the right version of the cells. The number of problems that occurred during NADS testing as a result of this decision along with all the lost time and productivity decisively proved the value of the original paradigm, which uses a single communication cell table.

For the NADS, scheduling within the main host was conveniently provided by the built-in operating system facilities of a frequency based scheduler (FBS). Scheduling remote hosts is implemented by generating remote interrupts through the Scramnet® boards.

More recently, the Concurrent Computer at NADS was replaced with a PC running Windows XP. The PC reads the scheduling file and generates an appropriate timed tick signal that is propagated through Scramnet® interrupts to the remaining hosts in the simulation. Since the PCI bus is fast enough to allow all data to be mapped on the Scramnet®, the duplicate communication cell tables have been eliminated, making integration and testing significantly easier. Interestingly enough, neither the IDS nor the NADS as initially delivered synchronized the scheduling with the Image Generator (IG). That caused occasional jitter when the drift between the IG clock and the scheduler clock exceeded the frame boundary, but the effect was never noticed due to IG eye point extrapolation, among other reasons. The PC implementation of the scheduler can synchronize with the IG sync signal by replacing the polling loop on the timer with an interrupt generator triggered by the IG output vertical retrace. Final testing of this capability is ongoing.

One key difference between the IDS and NADS is the notion of states. The IDS utilized a simple set of states that were communicated to each subsystem during the beginning of each execution frame. Three major states were provided: OFF, PAUSE, and RUN. A subsystem that performed some lengthy calculation during initialization would simply not call the blocking function, thus indicating it was busy. The operator would control the states through a simple command line interface program. The NADS utilizes a more sophisticated set of states and modes. The original IDS states are still available, but when in the RUN state there are approximately 30 modes. Each mode indicates a different phase that the simulator can be in, and even though only a small subset of these modes is encountered during routine operation, support must be provided for all of them. Example modes include

INITIALIZE, SETUP, PRE-POSITION, RUN, etc. Unlike the IDS, where subsystems implicitly acknowledged a new state by calling the blocking sync call, in the NADS, subsystems must explicitly acknowledge reaching the new mode by calling a separate function call.

ENHANCEMENTS TO THE REAL TIME FRAMEWORK

As NADS usage increases, the need to have access to low-cost part-task simulators that can be used for route familiarization, scenario development and testing, and post experiment review has also increased. However, the effort required to port the real-time framework on PCs has been exceeding the potential benefit. Recently, various simulator development projects provided the necessary critical mass to begin implementation of the real-time framework on commodity PCs. Since cost is a significant factor, the use of PCs running Windows 2000/XP® would be the platform of choice. At the same time, it was desirable to address various shortcomings of the framework that have been identified after its use on both the IDS and the NADS. These enhancements address the following areas:

Code Complexity and Quality

The original IDS framework interface was procedural and rather simple. The changes made to the NADS implementation, even though they followed the same paradigm, significantly complicated the interface, primarily because of the proliferation of the multiple states and modes, as well as the duplication of the communication cells. Furthermore, many decisions made by the NADS implementers were driven by short-term problem fixes at the detriment of consistency and code quality. Many of the internal symbols and data structures were visible at the subsystem level, and deadline-driven developers started using internal functions in an ad-hoc manner. An attempt to provide a C++ class interface made matters even worse because there was no coherent object-oriented design, but rather a simple grouping of functions into classes with a lot of dependencies and unrelated functions spread across class hierarchies.

The goal of the enhanced framework specification whose implementation is described in this paper is to provide a clean and coherent object-oriented interface that completely separates implementation details from the interface. Reducing the number of states and modes to make implementation of new subsystems simpler is now possible because the stringent safety requirements do not apply to systems that do not utilize a motion system such as the one used at the NADS.

Hardware Cost

One goal of the enhanced framework is to not depend on expensive hardware for inter-host communication. Because hardware details are not visible at the interface level, using readily available hardware affects only the implementation.

After some initial experimentation, it was decided that Ethernet networks have the required bandwidth and latency for real-time communication among hosts in a local and isolated network. Given that the granularity of the scheduler is in the millisecond range, Ethernet networks can be effectively used for propagating the scheduling tick as well as for transferring data through multicast.

Enhanced capabilities

Through the use of this framework for actual high-fidelity simulators, we have identified several key enhancements that can drastically simplify the complexity involved in putting together simulations based on the framework. These improvements are described below:

True object-oriented interface

Even though NADS compatibility was an important feature of the framework, it was decided that strict adherence to low-level code interface may be detrimental to overall performance and code quality. As a result, NADS compatibility was achieved primarily at the subsystem functionality level, not at a full source code level. This allowed the development of a new coherent object-oriented interface based on class inheritance, where each subsystem is implemented as a single class derived from a base class that implements all the functionality associated with the interface.

Strong typing

The original procedural implementation utilized untyped buffers whose appropriate sizing was the responsibility of the developer. There was no way to verify that buffers were large enough or typed according to the interface specification. The new interface provides facilities to verify type matching as well as proper sizing of buffers.

Elimination of system startup dependencies

Given some decisions associated with the design of the bootstrap code, it was necessary to start the scheduler process before starting the subsystems. Also, subsystems on remote hosts should be started before subsystems on the main host. The new implementation eliminates many of these dependencies, allowing arbitrary startup order for the various framework components. Subsystem startup is completely automatic so there is no need to manually start the subsystems.

Early flush of output data

The framework automatically reads input data cells before dispatching the subsystem computation function and writes the output cells when the subsystem completes its execution frame. However, this can be problematic in situations where the execution time of a subsystem varies through the simulation, even though part of its output data is ready early in the execution phase. For example, consider subsystem A producing output cells C1 and C2. C1 is ready after a short and fixed duration computation that takes place before a longer and variable computation responsible for producing C2. In the original framework, there was no way of writing C1 out before the end of the computation involved in producing C2. While micromanaging the time where each cell is written out is not the intent of this capability, there are cases, such as when C1 may be part of the transport delay chain, when flushing out certain data cells before the end of the computation is desirable. The new framework provides facilities for subsystems to selectively flush out certain cells before the current frame's execution has completed.

Figure 4 illustrates the effect of the early flush capability, which is indicated with the smaller arrow. Note how data written by the early flush will always be available for the next iteration of subsystem B, no matter how long subsystem A takes.

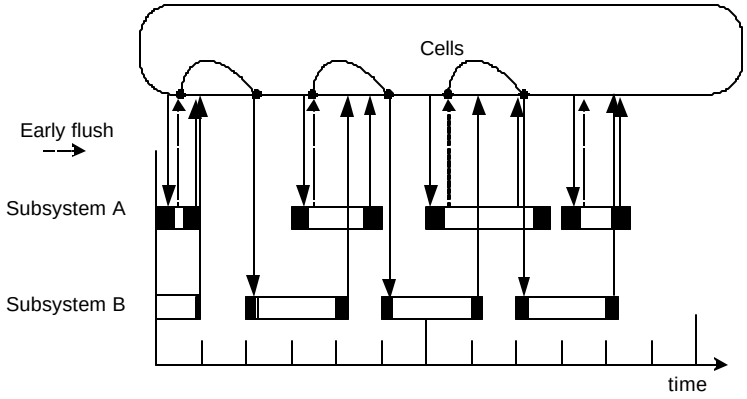

FIGURE 4 Illustration of the effect of an early flush of output data.

Elimination of race conditions during state transitions

As was painfully discovered by the NADS implementers, one problem associated with concurrent processes using state machines is the potential for race conditions when data flow is not explicitly taken into account when defining the states. For example, consider the SETUP state during which subsystems are supposed to perform initialization specific to the particular scenario. If subsystem A is meant to produce some configuration data that is to be used by subsystem B, and both activities are to be done during the SETUP state, there is a race condition upon the arrival of that state. During NADS integration, the developers solved these race conditions by creating numerous sub-state variables and writing a lot of similar-looking code spread across all subsystems that exhibited the problem. One can argue that this represents faulty design, rather than an inherent issue with state machines. This is true in the sense that adding a new state can easily solve the problem. For example, the SETUP state can be split in two states,

SETUP1 and SETUP2. Subsystem A is supposed to produce the configuration data during SETUP1 and subsystem B can read that data during SETUP2. This was not an option at the NADS because the state/mode design was decided early in the project phase and was ingrained in the design of all subsystems. Even if adding new states is possible, it may not be desirable because we want to control proliferation of states. The approach utilized in the enhanced framework is the addition of a new communication scheme based on a client-server paradigm that is meant to be used during non-real-time operation. It involves a centralized server that is responsible for supporting all non-real-time communication. Since real-time performance is not required, the server can solve all race conditions by properly serializing access to the various data items without the subsystems having to worry about implementing. Subsystems that require non-real-time data use the new facilities rather than the real-time cell paradigm.

IMPLEMENTATION OVERVIEW

The PC implementation of the enhanced framework utilizes Ethernet networking for communicating among hosts. Figure 5 illustrates the block diagram for the implementation. A simulation involves one PC acting as the main coordinator and one or more simulation hosts. For simplicity, Figure 5 shows a single host, although any number of hosts can be used.

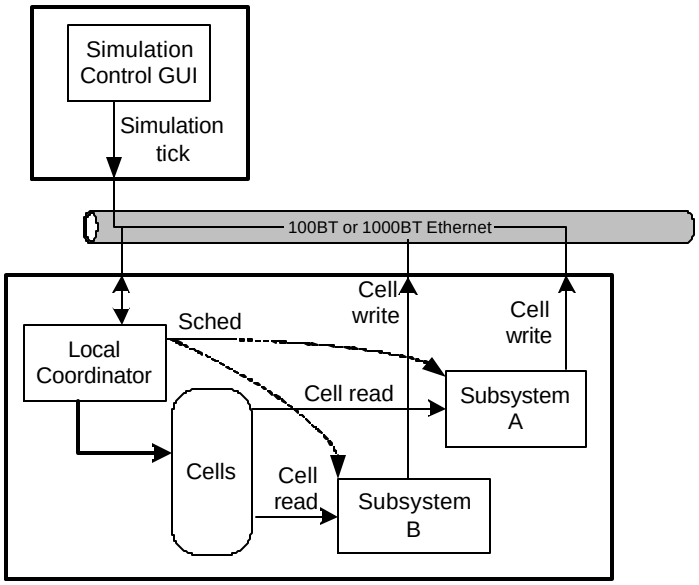

FIGURE 5 Framework implementation block diagram.

The main coordinator runs the simulation control Graphical User Interface (GUI). This program is responsible for the following activities:

- Managing simulation configurations, users, scenarios, and other necessary data objects
- Allowing the user to start/stop the simulation
- Transitioning between various operating modes (normal simulation or replay)
- Transferring any necessary configuration files to the simulation hosts
- Monitoring the health of all subsystems and hosts
- Producing the simulation tick used to define the minor frame of the simulation

To maximize the system's user friendliness, very few of these activities are explicitly specified by the user; rather, they take place transparently as the user controls the simulation.

Each simulation host contains a Local Coordinator (LC) process that is responsible for the following activities:

- Starting or terminating the subsystem processes as requested by the main GUI
- Maintaining a shared memory area that provides the cell storage area for local subsystems
- Receiving network commands defining the state of the simulation and communicating these to the local subsystems
- Monitoring the health of local subsystems and reporting the status back to the main GUI
- Receiving data transfer commands and writing the cell data to the local table

Typically, the main GUI and all the LCs need to be started on each host. After that, the user can pick the simulation to run and simply click a few commands to start it. At the appropriate times, the GUI sends commands to the LCs to download any configuration files or start the local subsystems. Once subsystems have successfully initialized, the GUI allows the user to start the simulation.

To ease implementation of the networking code, the Network Data Delivery System (NDDS) library [4] is used. This middleware library provides a flexible interface modeled after the Publish & Subscribe (P&S) model of communication. This is a best-effort model where data is published and received by any entity that has subscribed to it. The library allows transparent use of multi-cast transmissions, which drastically reduces the network load. In addition, the library automatically packages multiple user level messages into single Ethernet packets, thus maximizing the bandwidth and minimizing latency. As an added benefit, the library also provides a client-server mode of communication that is used during the initialization phase.

At runtime, the GUI program is responsible for generating a publication that represents the scheduling tick. The LCs subscribe to the tick. Each time the tick packet (or *issue* in P&S nomenclature) is received by an LC, the LC calculates whether any of the local subsystems should execute. If so, a quick check on the subsystem status is made. If it is executing, then the subsystem has exceeded its allocated time, in which case the overrun counter is incremented and no further action is taken. If the subsystem is blocked, then a Windows inter-process communication message is used to wake it up. The subsystem then reads its input cells, performs its computation for one frame, writes out its output cells, and blocks again.

Data communication is implemented transparently within the real-time simulation framework. Specifically, unless using the “early flush” primitive, subsystems don’t have to explicitly issue any read or write statements. They only have to register their input/output cells during initialization. The framework, on the other hand, is responsible for implementing this read/write capability. This is achieved in a non-symmetric manner, in the sense that reading is done by directly accessing the local cell storage area in shared memory, but writing is done by sending out a publication. More specifically, as subsystems register their output variables, matching publications are created for each output cell. These publications are intercepted by the LCs on all hosts, which then subscribe to them. When a subsystem completes its execution for a frame, all output data is published through the NDDS framework. The LCs, receive these issues and copy the data to their local cell storage area, where they are available for local subsystems to read. Input cells are read directly from the local cell storage, which is stored in a shared memory segment on each host.

During the design of the system, there was some concern about performance. In practice, performance has been more than adequate for building simulations, even when the time tick is in the range of a few milliseconds. Contributing factors to this performance level include the use of multi-casting that allows simultaneous transfer of data to many hosts, as well as the performance of the PCI bus in PCs that allows the in-memory transfer of data that is nearly negligible.

The API

This section provides a summary of the Applications Programming Interface (API) presented to subsystems. As mentioned earlier, the API is implemented in the form of a C++ class from which subsystem implementations inherit the frameworks’ functionality. A few of the details have been eliminated for clarity, but the core of the API is provided below.

```
Class SubsysBase {
                  SubsysBase(int id);
      int         GetId(void);
      void        MainSimLoop(void);
      float       GetFrequency(void);
      float       GetPeriod(void);
      float       GetRunTime(void);
      int         GetRunFrame(void);
      unsigned int GetFrameCount(void);
      void        AckStateMode(int StateMode);
      virtual int  InitSubsystem(void)         = 0;
      virtual int  RunSubsystem(int StateMode) = 0;
      virtual int  Shutdown(void)              = 0;

      int         RegInpElem(string & Name, int* pBuff, int size);
      int         RegInpElem(string & Name, float* pBuff, int size);
      int         RegOutElem(string & Name, const int* pBuff, int size);
      int         RegOutElem(string & Name, const float* pBuff, int size);
      int         FlushElems(int which[], int len)
}
```

Typically, a subsystem would derive a class from SubsysBase and specify the subsystem's identifier in the constructor. The majority of a subsystem is implemented in the InitSubsystem(), RunSubsystem(), and Shutdown() functions.

After instancing the class, the subsystem calls MainSimLoop(), at which point all control is passed to the simulation framework. The InitSubsystem() function will be called once during initialization. It is expected that the subsystem will register its input and output cells at that time. The RegInpElem() and RegOutElem() functions allow registering of input and output elements, respectively. The functions are overloaded to provide a flexible API for all data types. Cells (and their *elements*) are referred to by name, but integer handles are returned.

At runtime, the framework will periodically call the RunSubsystem() function. The sole argument is the intended state of the simulation. The subsystem then can proceed with its execution and eventually acknowledge transition to the requested state/mode by calling the AckStateMode() function. The framework guarantees that before the RunSubsystem() function is called, all input elements have been copied from the local host's cell table to the subsystem's memory specified in the registration functions. Similarly, all output cells will be broadcast upon return of the RunSubsystem() function. The FlushElems() function implements the "early flush" capabilities. Calling this function forces the elements whose identifiers are contained in the argument list to be broadcast at the time the function is called.

Once the simulation is commanded to terminate, the Shutdown function is called, allowing the subsystem a chance to clean up any data structures or free hardware resources.

The remaining functions listed in the API are utility functions that do not have to be used, but are available if needed.

PERFORMANCE EVALUATION

Overview

Upon development of the initial version of the library, a series of tests was devised to measure the performance of the system and assess its applicability for real-time driving simulation applications. Typically, driving simulation applications involve subsystems to perform vehicle dynamics, scenario control, scene rendering, audio generation, cab interface, data collection, etc. Pertinent to the frameworks' applicability for such applications is the expected execution periods of these subsystems. Note that we were not interested in each subsystem's requirements for internal execution rates, but rather the requirement for the rate at which each subsystem must communicate with the remaining subsystems. For example, a dynamics model may execute at a rate of 1000 times a second; however, it

may only need to sample the control inputs at 100 times a second. For purposes of evaluating the determinism of the real-time framework, an execution frequency of 100 Hz would then be considered, with the dynamics model executing multiple iterations for each time it is scheduled by the real time system. Generally, with rendering rarely taking place at faster than 60 Hz, most subsystems can communicate at rates that are small multiples of that frequency; for example, the dynamics at 120 or 240 Hz, scenario control at 60 or even 30 Hz, etc. With the above in mind, the tests focused on a minor frame of no less than 4 milliseconds, or a maximum frequency of 250 Hz.

Generating the Tick signal

A central issue in establishing a deterministic simulation is the generation of a relatively jitter-free tick signal. In this case, the signal appears in the form of a network packet (or *issue*) that must be generated by the main coordinator GUI. Even though there are multitudes of methods that can be used to generate such a periodic signal, two specific methods were investigated. One is to use an external hardware source (such as a waveform generator or the monitor vertical sync signal). Another way is to execute a timing loop sending the sync signal each time the necessary interval has expired. The advantages of the former method include the ability to synchronize the whole simulation with the image generator; however, some of the disadvantages include the potential for jitter due to Windows interrupt dispatch latency. There is also the issue of needing to integrate the additional hardware in the system. The latter method does not require any hardware; however, there are two key problems. One is the drift of the IG clock relative to the clock on the PC that generates the sync signal; this generally appears as a periodic jitter, which may or may not be noticeable, and may even be correctable with some form of interpolation. A second problem with the latter method is the lack of an interruptible timer with high enough resolution to provide periodic interrupts with a period of a few milliseconds. The Windows Sleep() function is supposed to suspend a process for the specified number of milliseconds; however, it is rather inaccurate. Figure 6 illustrates some plots of the interval that the Sleep() function actually gave up control for various arguments. The Pentium high-resolution timer was used to measure the actual time between when the function was invoked and when it returned control to the calling program. The following loop is representative of the code segment that was used to obtain these results, with the variable "period" holding the intended number of milliseconds that the process wishes to block.

```
for (i=0; i<1000; i++) {
      StartHighResMeasurement();
      Sleep(period);
      elapsedTime[i] = EndHighResMeasurement();
}
```

The simple experiments were done on a 2.0GHz PC running Windows 2000, although it is unlikely that the actual processor speed would make an appreciable difference. There was no other load on the system at that time. It is readily apparent when looking at these plots that there are two problems. The first is the variation of the actual delay and the second is the absolute accuracy. In fact, the variation gets worse if some load is added to the system while the timing measurements are taking place. To evaluate the effect of load, a second test was run. In this test one application was started while the measurement loop was executing. Starting an application provides a variety of activities involving disk access, memory accesses, and operating system calls. Figure 7 illustrates the result in this case. As is evident, the variation in this case is completely unacceptable.

To address the variation, one can modify the default process priorities. Windows NT and later provides two primitives that control how a thread is scheduled. The process class, which can be controlled by the *SetPriorityClass* system call, modifies the base priority of all threads in a process. Individual threads can have their priority controlled as well by using the *SetThreadPriority* system call. The highest priority thread can be achieved by setting the process class to real time (*REALTIME_PRIORITY_CLASS*) and then setting the thread priority to "time critical" (*THREAD_PRIORITY_TIME_CRITICAL*). It is important to note that a thread running under such high priority will preempt even operating system calls. Any coding error that leads to an endless loop can easily starve every other thread, including threads that process keyboard and mouse input, thus necessitating a cold reboot. Figure 8 illustrates the same test taking place under similar load conditions, but with the highest possible thread and process priorities.

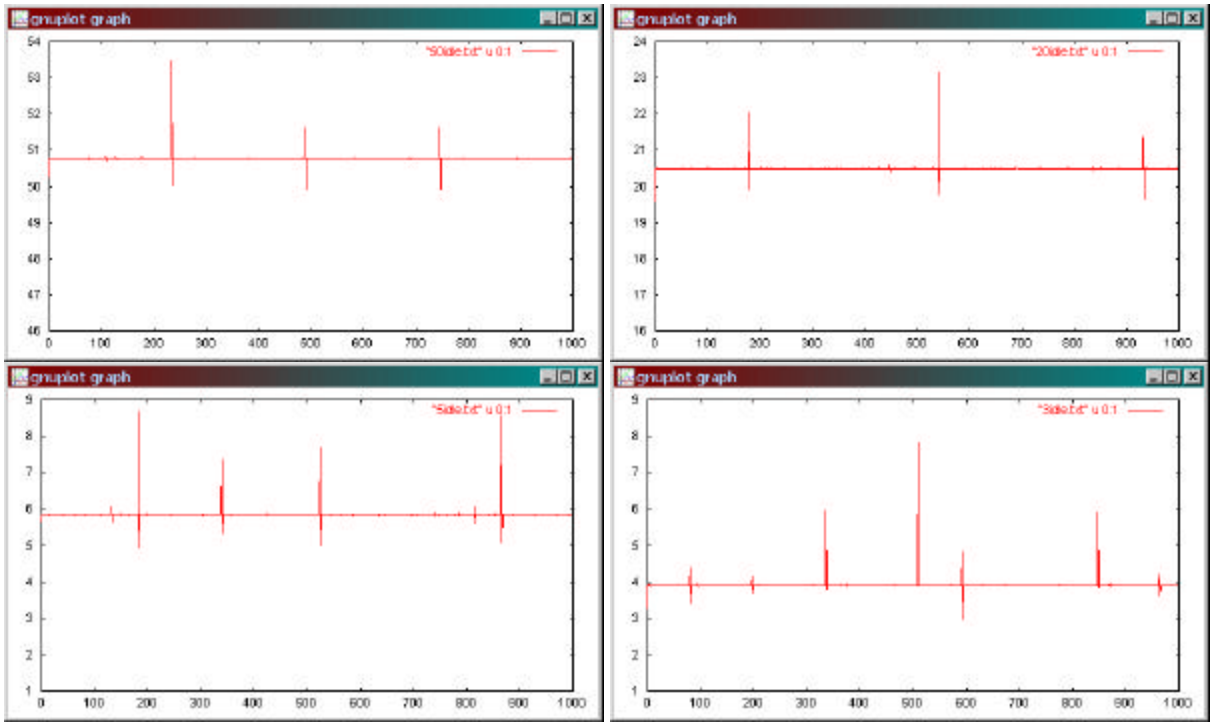

FIGURE 6 Tick interval using Sleep() function for 50, 20, 5 and 4 milliseconds.

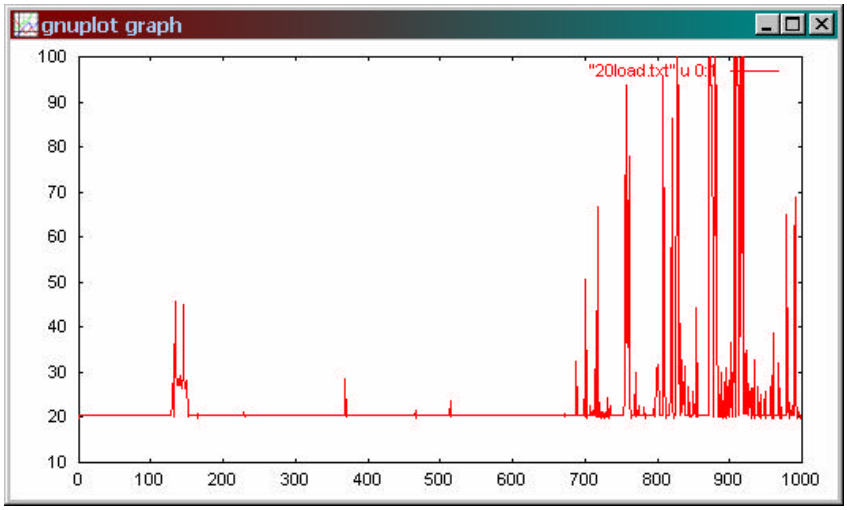

FIGURE 7 Tick interval using Sleep under the presence of load.

As is apparent in Figure 8, the variation has largely diminished to acceptable levels, at least for the typical real-time granularity involved in driving simulation applications. However, the absolute accuracy of the interval is not acceptable. In the left subfigure, the intended interval was 20 milliseconds and in the right subfigure the intended interval was 3 milliseconds. This can easily be explained by the default granularity of the Windows scheduler.

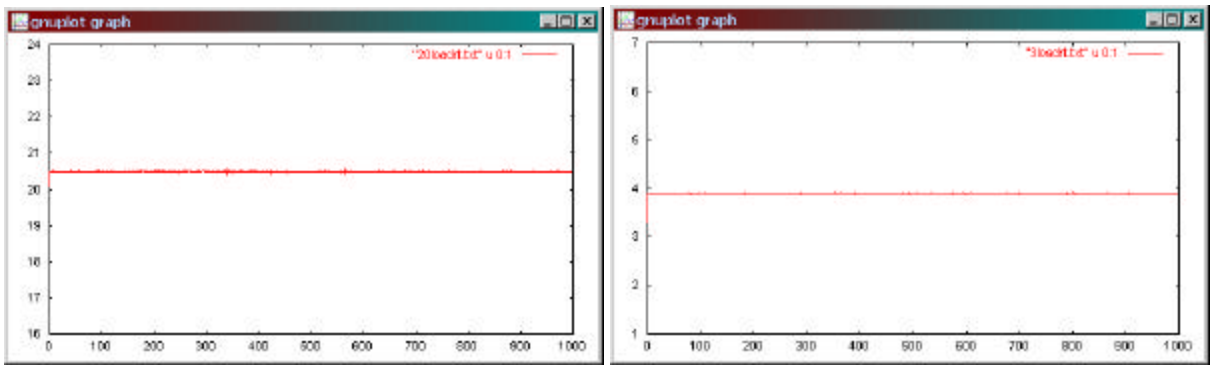

FIGURE 8 Tick interval using Sleep() under real-time mode.

To address this last issue, a technique that involves a hybrid of busy polling and blocking using Sleep() was used. Specifically, the following code structure was used to run another small experiment:

```
for (i=0; i<1000; i++) {
      StartHighResMeasurement();
      Sleep(period/2);
      while ( EndHighResMeasurement() < period )
            ;
      elapsedTime[i] = EndHighResMeasurement();
}
```

This approach has the advantage of releasing the CPU enough to provide acceptable interactivity for the user, yet using polling to fine tune the exact elapsed time. Figure 9 illustrates the result of running this code for 3 and 20 milliseconds, respectively, under external process load. The y axis units in this case are milliseconds. Note the narrower y axis range in these plots as compared to the ranges used in Figures 6-8. A single jitter in the right subfigure is about 0.3 milliseconds, which is within acceptable limits.

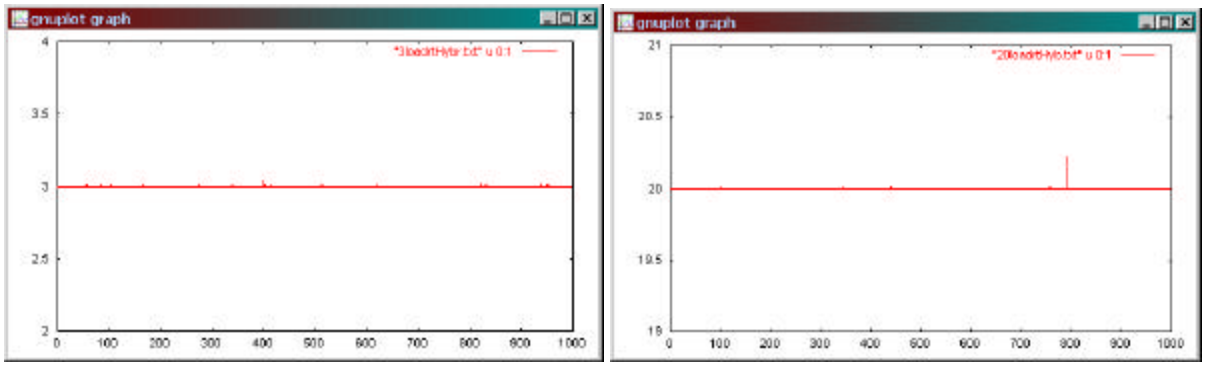

FIGURE 9 Tick interval using the hybrid polling/Sleep() approach.

Having a method by which elapsed interval can be accurately and deterministically measured allows the generation of the master tick signal using only software, if that is so desired. In cases where the tick signal was produced by an external interrupt, results similar to Figure 9 were obtained. The following code segment illustrates a summarized version of the code used to broadcast the scheduling tick to all hosts in the simulation. The code sends the tick signal and then uses the hybrid method for producing a delay to wait until the next period arrives.

```
while ( SimulationActive ) {
      StartHighResMeasurement();
      BroadcastTickSignal();
      y = EndHighResMeasurement();
      timeLeft = x – y;
      Sleep(timeLeft/2);
      while ( EndHighResMeasurement() < timeLeft )
            ;
}
```

Signal Propagation across the Network

Having obtained a consistent method of producing the master tick signal, the next concern was how the propagation of the tick signal through the network and through the LC would affect the determinism of the system. In general, TCP/IP and UDP communication over Ethernet is not considered real-time communication in the sense that packets may get lost or any network collisions would cause re-transmits, which in turn can introduce arbitrary delays.

Figure 10 illustrates two graphs of the measured interval time between subsystem invocations for the subsystems that reside on different hosts. In this case, the vertical axis is in microseconds, and the intended period was 33333 microseconds (30 Hz). The three hosts that comprised the system for this experiment were on a private network, with the exception of the main host running the GUI that contained two Network Interface Cards (NICs), one connected to the private network and one connected to the University's internal network. The private network was implemented by using a commodity 10BT hub. Each of the two subsystems whose scheduling signal interarrival spacing is shown in Figure 10 resided on its own host. Note that even though the jitter is higher than what is obtained by the tick generation code, the overall deviation is acceptable for the required applications. In each plot whose duration is approximately 30 seconds, there are one or two frames where the deviation from the intended interval was as high as 1.7 milliseconds (approximately 5% variation). It is unclear whether such variations can actually disrupt the data flow across the system, but they require further attention. Possible explanations for this behavior include Windows service processes that utilize the network at the same time coupled with other self-initiated system activity. Even though no other user processes were running on these systems during the tests, there was no specific effort to "sanitize" each of the PCs, for example by shutting down unnecessary services or uninstalling regular applications.

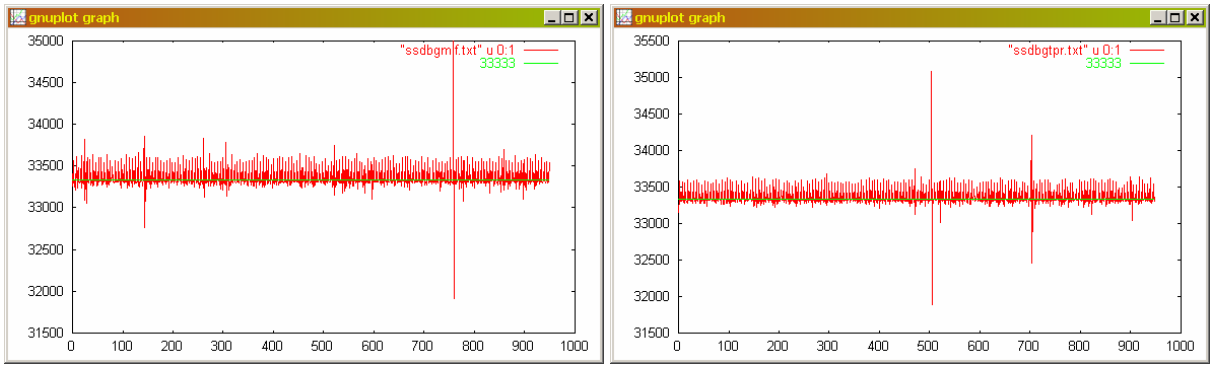

FIGURE 10 Invocation interval across the network.

Total System Stress-Test Result

To determine whether the small amount of jitter would disrupt the data flow across the system, a more comprehensive stress test was devised. Figure 11 shows a block diagram of the test setup. One PC was used to run the GUI, which also generates the tick signal. Two more PCs were used, hosting a total of 3 test subsystems. These subsystems performed no useful computation other than reading the input cells, performing a trivial computation, and storing the result in their output cells. The cell configuration file contained three identically structured data sets specifically designed to be used with the stress test. For simplicity, let us refer to the three identically structured cells as A, B, and C. Each cell contained a variety of data elements covering all data types and various size arrays.

Specifically, there were one integer scalar and two integer arrays of 12 and 200 items respectively, one short word scalar and two short word arrays of sizes 5 and 200, one floating point scalar and two floating point arrays of sizes 64 and 256, one double floating point scalar and one double floating point array with 256 items, and three character arrays with lengths of 4, 8, and 256 bytes. A total number of 4872 bytes was included in each of the three cells. The variety of data types and array cardinality is meant to exercise all aspects of the code, and the NDDS library that is responsible for performing the network communication as well as the data packing and unpacking.

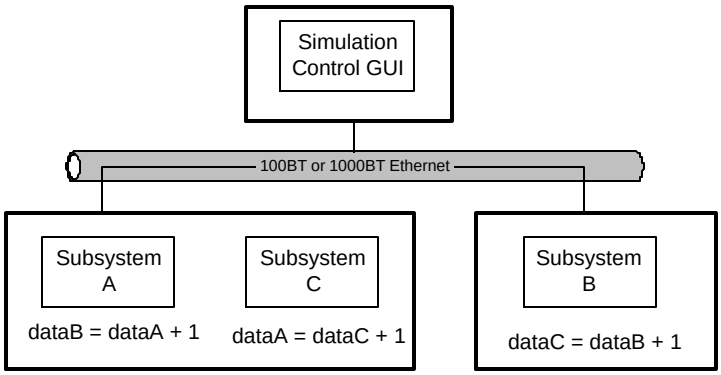

FIGURE 11 Stress test setup.

The computation performed by each subsystem involves adding the value of 1 to each data item contained in a cell. Subsystem A was coded so it reads cell A, increments by one, and writes output to cell B. Similarly, subsystem B reads from cell B and writes to cell C. Finally, subsystem C reads from cell C and writes to cell A. One can easily see that this creates a circular data flow. As long as all data is transferred properly and on time, the values in the cells should increment once per simulation tick.

The schedule used in the test is shown in Figure 12. Subsystem A runs on the same host as C but a different host than B. To maintain proper dataflow, the real-time framework must deliver the data between the two hosts at each simulation tick.

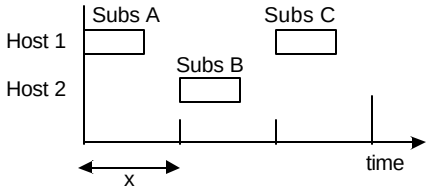

Figure 12 Stress test schedule.

The test involved varying the value of x (tick interval) and, for each value, running the simulation for approximately 2 minutes. At the end of the simulation, the number of ticks for which the simulation was in the RUN state was compared with the final value stored in the cells. Because changing state/modes does not necessarily happen on major frame boundaries, the largest value from among any of the three cells was used. The number of missed frames was then recorded. Table 1 provides a summary of the results, including the effective bandwidth utilized by the system.

TABLE 1 Missed frames versus tick duration.

Tick Duration (mSec)	Missed Frames	Bandwidth	Missed Frames (Test 2)
6.67	17	728 Kb/sec	1
8.33	15	580 Kb/sec	1
11.11	5	438 Kb/sec	1
12.35	3	392 Kb/sec	1
13.33	1	364 Kb/sec	1
16.67	1	290 Kb/sec	1

The effective bandwidth takes into account that during the schedule shown in Figure 12, 4848 bytes of useful payload have to be transferred from one host to the remaining ones in the simulation. The bandwidth does not contain any additional book-keeping information that has to be transferred among hosts. It also excludes the packet that conveys the simulation tick. The results shown in Table 1 were obtained on a 10 Mbit/sec channel using an Ethernet hub.

A second test was also run. In this case, subsystems A and B resided on a single host. This necessitated a single transfer per two simulation ticks, in effect reducing the bandwidth in half. As shown in Table 1, a single dropped frame was measured in this case.

It is interesting to note that most of the missed frames took place at the beginning of the simulation. Clearly, even with tick periods at 6.67 milliseconds, the dropped frames are very low, and once the tick period exceeds 12 milliseconds, the dropped frames become negligible. One would also anticipate that for the 6.67 millisecond tick, the limiting factor was the network bandwidth given that a 10Mbit/sec network was used. The NDDS library coalesces multiple issues into single network packets, so that no matter how many individual data items are managed, the actual network transfers take place using the largest, and thus most efficient, network packets possible. In an actual simulator environment, 100BT or 1000BT links would be used, which would drastically increase the available bandwidth.

Finally, because the network transfers utilize multi-cast, it is reasonable to assume that the system will not be negatively affected by the use of a larger number of hosts, as is more likely to be the case in an actual simulator.

CONCLUSION

The paper described a framework for integrating real-time systems focused on the real-time performance envelop of driving simulator applications. A historical overview of a similar framework was presented along with the rationale for various improvements and enhancements incorporated into the latest version of the frame work. In addition, an implementation of this framework specifically targeting PC hardware running Windows ® NT/2000/XP was described, and its performance was evaluated when using local area Ethernet networks for inter-host communication. The feasibility of using Windows ® NT/2000/XP was investigated for real-time applications whose granularity matches typical driving simulation applications. The communication layer is built on top of Real Time Innovations Corporation's NDDS middleware library, which facilitates the implementation of efficient, multi-cast enabled communications among simulator hosts, while allowing the real-time framework to provide a simple yet flexible interface for subsystem development.

The performance of the framework was found to be quite adequate based on typical driving simulation application requirements.

ACKNOWLEDGMENTS

This work has been partially supported by IDC Simulation Systems, the National Science Foundation Industry/University Cooperative Research Center for Virtual Proving Ground, and the Real Time Innovations Corporation's University Program.

REFERENCES

1. Kuhl, J. G., Papelis, Y. E. A Software Architecture for Operator-in-the-Loop Simulator Design and Integration. *Workshop on Parallel and Distributed Real-Time Systems*, April 1993, Newport Beach, CA, pp. 117-126.

2. Freeman, J. S., Watson, G., Papelis, Y. E., Tayyab, A., Romano, R. A., Kuhl, J. G., The Iowa Driving Simulator: An Implementation and Application Overview, *SAE International Congress and Exposition*, February 27-March 2, 1995, Detroit, Mich, pp. 81-90.

3. Brewer, H. K., National Advanced Driving Simulator. Paper presented at the Global Automotive Safety Conference, Society of Plastics Engineers, February 5, 2001.

4. Real Time Innovations Corporation. The network data delivery service. White paper, http://www.rti.com.