Distributed Real-Time Embedded Applications using Off-the-shelf Components: Experiences Building a Flight Simulator

Sundeep Kapila, Krithi Ramamritham, K. Sudhakar

Abstract— **Flight simulators are motivated primarily by financial and physical constraints in using the actual system for pilot training. They are excellent examples of resource intensive distributed embedded applications. They are also highly demanding in their timing requirements. This paper reports on the design, implementation, and evaluation of a research flight simulator developed using off-the-shelf components: operating system, flight dynamics software, as well as networking and communication software and hardware. While our overall experience was quite positive, we discuss how we overcame some of the difficulties we encountered in the communications area as well as in measuring pilot-input-to-response times.**

Keywords— **Real-time systems, flight simulator, Real-Time Linux, COTS.**

1. Introduction

Flight Simulators are excellent examples of resource intensive embedded applications. They are desktop applications, but come as an embedded package, as for example, video games, and are often made up of distributed components. Future flight simulators are expected to be pack as much of the cockpit as possible into lower-cost flight training devices [5]. These will use software to drive the virtual reality displays and offer the necessary functions without having the actual flight hardware. While the most cost effective way to achieve all this in a timely fashion is to use commercially available off-the-shelf software, hardware, and system components, a major challenge lies in satisfying the stringent timing constraints attached to flight simulator functions.

This paper reports on the design, implementation, and evaluation of a research flight simulator developed using off-the-shelf components: operating system, flight dynamics software, as well as networking and communication software and hardware. We report on the challenges faced and the solutions adopted to meet them. The design of the system has been validated by simulating numerous scenarios and shown to meet the performance requirements of the system.

This paper is organized as follows. Section 2 gives a brief overview of flight simulators and their performance requirements. It also explains the issues, including timing issues, that arise in the design of the system and elaborates on a possible design of the system. Section 3 focuses on implementation details as they pertain to (a) the use of RTLinux and (b) communication between nodes. Some challenges in, and solutions for, timing measurements, are detailed here. Section 4 discusses the experimental setup and analyses the results. Finally, Section 5 summarizes our experiences and lessons learned.

Sundeep Kapila and Krithi Ramamritham are with the Department of Computer Science and Engineering, K. Sudhakar is with the Department of Aerospace Engineering, Indian Institute of Technology, Bombay, India. (email: sundeepkapila@yahoo.com, krithi@cse.iitb.ac.in, sudhakar@aero.iitb.ac.in)

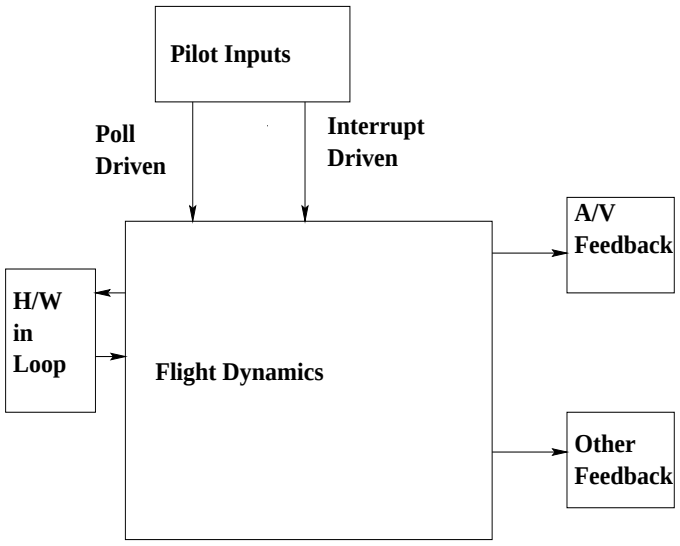

Fig. 1. Architecture of a Flight Simulator

2. Flight Simulator

Flight simulators are motivated primarily by financial and physical constraints in using the actual system for pilot training [3], [4]. Besides training, flight simulators are also used for research purposes to test the authenticity of various flight dynamic and control models used for modeling aircraft motion.

2.1 Architecture of the System

The basic components [see Figure 1] of a flight simulator are -

- *Pilot Inputs* - This is responsible for taking input from the pilot. There are primarily two types of inputs, continuous inputs, like joystick inputs and interrupt driven inputs, like inputs from toggle switches.
- *Flight Dynamics Engine* - This is responsible for taking the current state of the system and the pilot inputs and evaluating the new state of the system. This typically involves solving a system of ordinary differential equations of

order twelve. This can also involve interacting with hardware.

• *Feedback System* - This includes the audio and visual feedback (A/V feedback) given to the pilot as per the current state of the system. This might also involve feedback to other devices like the motion platform.

In a typical flight simulator, the components would be on different nodes forming a distributed system. These components communicate with each other by different means including network, com port and Data Acquisition Cards (DAQ cards), for example, the latter is used for joystick input via the game port.

The motivation behind having a distributed architecture is the ability to independently develop the modules, each of which requires different specialization. Such an architecture also facilitates extending this system to multiple user and aircraft simulations.

2.2 Our Basic Design

Our basic infrastructure involves two nodes, known henceforth as Client and Server, besides components that correspond to the hardware in the loop, and the devices for input and output. The Client represents the node from/through which the pilots inputs are read and the A/V feedback is given to the pilot. The Server is the Flight Dynamics module, which evolves the time history of the aircraft based on pilot inputs.

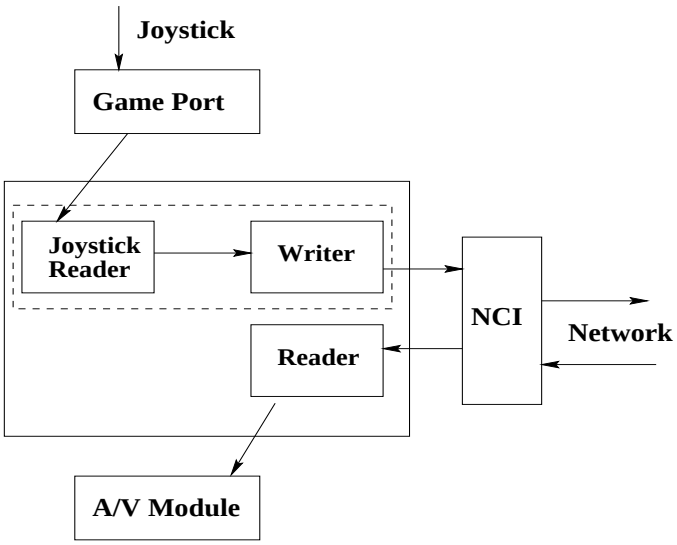

Fig. 2. Client Architecture

There are essentially two threads running on the Client [See Figure 2].

• A thread that reads pilot inputs like joystick inputs (from the "Game Port" buffer) and transmits the input stored in the buffer over the network to the Server. (So, the "Joystick Reader" and "Writer" are implemented as a single thread.)

• A thread that polls the network card interface (NCI) and uses the received data to display the new state of the aircraft through the A/V module.

There are five threads on the Server [See Figure 3].

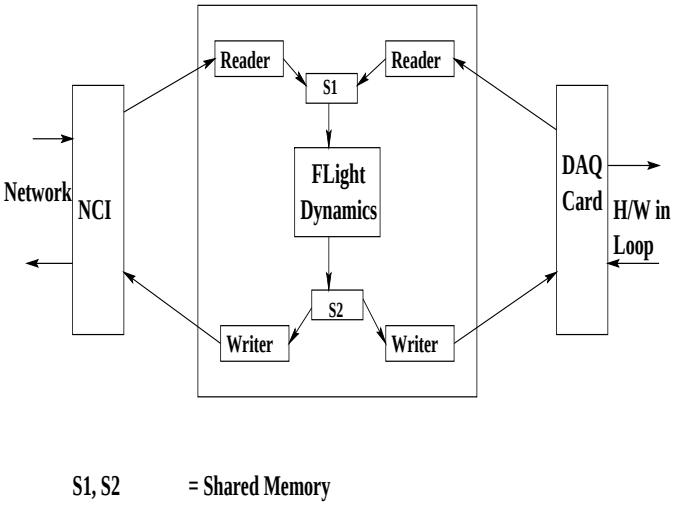

Fig. 3. Server Architecture

• One thread each for reading data regularly from (a) the H/W in the loop and (b) the network card and writing it to a shared memory location (S1).

• One thread each for regularly reading data from a shared memory location (S2) and sending it through to (a) the network and (b) to the H/W in the loop.

• A thread for reading input from S1 and computing the new state of the system and writing it to S2. This is the Flight Dynamics module of the system.

2.3 Timing Issues

In this section, we examine the application specific timing constraints and also state how they translate to timing requirements on the component (threads).

The timing requirements imposed by the application on the system are -

• Response to a given pilot input should occur within less than $150ms$ for commercial aircrafts and $100ms$ for fighter aircrafts[2].

• Continuous pilot inputs like joystick should be polled at rates greater than once every $16ms$ [6].

• The state of the aircraft (as calculated by the code which solves the equation of motion of the aircraft) is to be advanced at $12.5ms$ time steps.

As we shall see, the time periods of the different threads are chosen to meet the above timing requirements of the Flight Simulator.

• The Flight Dynamics thread has a time period of $12.5ms$. This thread involves floating point operations for solving differential equations. Hence, the execution time of the thread depends on the nature of pilot inputs. However, the new state calculated is made available only at the beginning of the next time period after $12.5ms$ as per requirements.

• The reader and writer threads on the Server should ideally run at high rates so as to minimize the response time. However, very small time periods lead to overload (as explained later), hence a time period of $4ms$ was used for them.

• The same reasoning as above holds for choosing the time

period of the reader thread on the Client. The rate at which the A/V module needs to receive data was also taken into account while choosing the value of $4ms$.

- The period of the writer on the Client relates to the rate at which the joystick is polled for pilot input. The value of this time period, as per requirements, should be less than $16ms$. We keep this period as a variable for evaluation purposes and test with values from 4 to 16 ms and study the effect on the overall pilot-input-to-response latency.

The above choice of time periods, ignoring scheduling and interrupt latencies (which are shown to be small given our implementation), ensures that

- The continuous pilot inputs like joystick are polled at rates greater than $16ms$ [6].
- The state of the aircraft, as calculated by the thread which solves the equation of motion of the aircraft, is advanced at $12.5ms$ time step.

The other timing requirement of bounded pilot-input-to-response latency needs to be ensured. We now present an analytical value for this bound.

The input data read and transmitted by the Client's writer thread flows through different threads in the following sequence - reader on the Server, flight dynamics thread on the Server, writer on the Server and then finally the reader on the Client. Each of the threads performs three actions - read data from a shared location or buffer, process it and then write it to a shared location. The maximum delay between data written in a shared location and same data being read, processed and passed on to another thread is equal to the time period of the thread and its execution time, except for the flight dynamics thread in which processed data is written at the beginning of the next cycle. Hence, the maximum delay for the flight dynamics thread is twice its time period. Hence the total maximum delay Δ encountered by data that leaves the Client end until the Server's response is ready for delivery to the A/V module is

$$\begin{aligned}\Delta \leq & \textit{time period of reader on server} \\ + \ & 2{*}\textit{time period of flight dynamics thread} \\ + \ & \textit{time period of writer on server} \\ + \ & \textit{time period of reader on client} \\ + \ & \textit{execution times of reader and writer on Server} \\ + \ & \textit{network delays}\end{aligned}$$

Since the execution times (of the readers and writers) and the network delays are ensured to be small – through careful implementation – compared to the time periods (they are of the order of 10's of μsecs in our system), we obtain, based on our chosen values for time periods,

$$\Delta \leq 37ms$$

Hence, we have a bound on the value of the latency experienced by an input generated by a Client until it is ready for delivery to the A/V unit. This is used in the next section to determine the overall end-to-end delay.

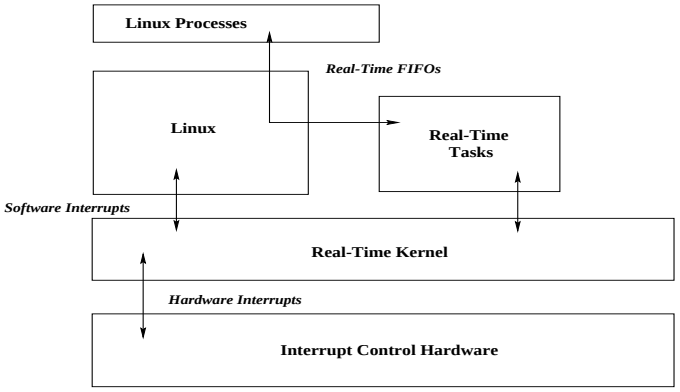

Fig. 4. Architecture of RTLinux

3. System Support

It is clear that in order to execute the operations of the flight simulator predictably, it is important that the overheads involved in the interactions among the flight simulator components are bounded in order to achieve the timing requirements of the Flight Simulator. During communication between the two nodes, besides the unpredictability due to transmission delays, there are other reasons for unpredictability:

- Unpredictability in interrupt latency of the network interrupt.
- Unpredictability in delays caused in the kernel due to the movement of a packet through the protocol stack.
- The delays in the Com Port and DAQ Card interactions can be unpredictable because of scheduling or interrupt latencies caused by the system.

Choosing the operating system and communication infrastructure that minimizes, if not eliminates, the above sources of unpredictability is an important issue in the design of system support for modern flight simulators.

In this section, we discuss how RTLinux was used to provide system support, in particular, to reduce interrupt latencies. Hence, this choice necessitated developing a device driver for the network interface card we had. The principles underlying the design of this device driver are then described. Finally, we discuss how we enhanced the timing measurement capability to cover events that span both the real-time and non real-time spaces.

3.1 Use of RTLinux for OS Support

It is well known that common-off-the-shelf (COTS) operating systems are not desirable platforms to run hard real-time tasks. However, using a familiar environment for development which has all the other functionalities of a COTS operating system is desirable. Hence, we decided to use RTLinux [1], as the operating system on which to build our application. It provides all the functionality of linux and also provides hard real-time support.

In RTLinux, the normal linux kernel [See Figure 4] is run as the lowest priority process. The real time processes get priority over the linux kernel and can preempt it. The RTLinux kernel layer lies between the hardware and the

linux kernel. It receives all the interrupts from the hardware and checks whether there is any real-time process waiting for the interrupt. If there is such a process, then it schedules it immediately, else it transfers the control of execution to the linux kernel. The real-time processes run in memory space which is reserved and isolated from the non-real time processes' memory space. Hence, context switching from non real-time mode to real-time mode is fast and predictable since no state has to be saved. Interrupt latencies in RTLinux are typically of the order of 10 *microseconds*. Real-time schedulers are supported as kernel modules. RTLinux is an open source operating system and provides a familiar threads programming environment.

The basic philosophy behind the design of RTLinux is that any real-time application can be segmented into two types of processes - small (in memory and resource usage) hard real-time processes and larger (in memory and resource usage) soft real-time processes. Applications designed in RTLinux have two components - real-time and non real-time. Real-time and non real-time processes communicate via real-time `fifo`s which are implemented as character devices. For each `fifo`, one can associate a handler which is run as a real-time process and is activated whenever something is written to a `fifo`. Non real-time processes access the real-time `fifo`s like files.

Another important aspect of the application development environment of RTLinux is the fact that the memory space allocated for real-time processes is fixed and there is no dynamic allocation of memory in the real-time space.

3.2 Supporting Communication between Components

Com Port and Data Acquisition (DAQ) Card interactions can be bounded since they involve reading data from particular ports. This can be done in real-time space by periodic threads. Moreover, there are real-time drivers for serial ports available with the standard distribution of RTLinux.

In case of network interactions, one can impose bounds on interrupt latencies by developing a real-time network driver. In order to bound the protocol stack delays, the interactions can be made at the ethernet protocol level and avoid the TCP/IP stack. Hence, we developed a real-time network card driver which allows communication at the ethernet level for a certain class of identified packets and normal communication for the rest.

The RTLinux kernel interacts with the hardware devices. The linux drivers run as non real-time interrupt handlers and hence are subject to latencies as detailed previously. If an application desires to access a particular device in real-time mode, then a real-time driver for RTLinux for that device is required. Network interaction is an important part of the implementation as detailed above. Hence, we need to access the network cards in real-time mode. Whereas drivers for tulip and 3com network cards for RTLinux are available, real-time drivers are not available for Real Tek network cards, the cards we had access to, even though they are widely used. So we developed the driver for RTLinux for Real Tek 8139 network cards.

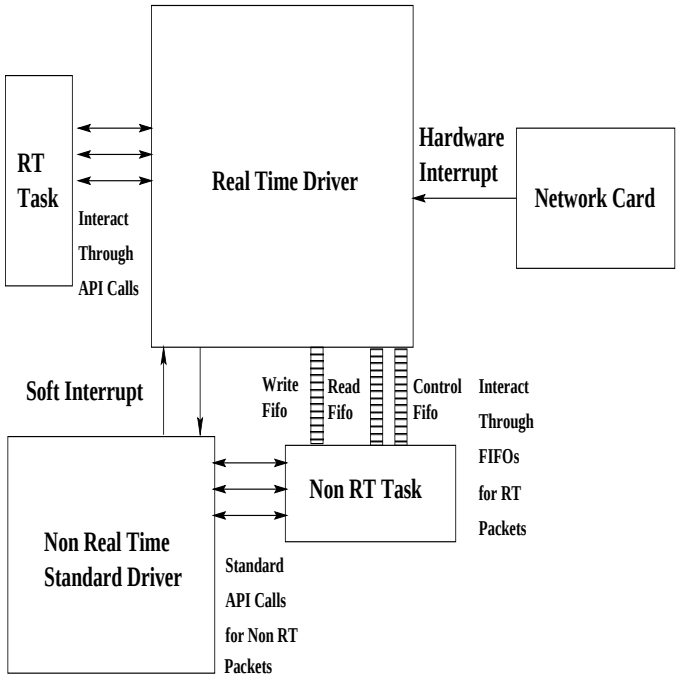

Fig. 5. Network Card Driver Architecture

The implementation of the driver required the design of a protocol for timely data transmission. In this protocol, a bit in the ethernet header is set or cleared to indicate whether the packet to be transmitted is a real-time packet or not. This bit is checked at the receiver to identify real-time packets. The architecture of the driver design is as shown in Figure 5. As seen in the figure, two drivers, one real-time and another non real-time coexist in the system. The interrupts from hardware are received by the real-time driver. Real-time packets are directly and immediately processed by the real-time driver while non real-time packets are initially buffered in ring-buffers and later processed by the non real-time driver when the linux kernel gets scheduled.

The interaction of the real-time driver with real-time tasks is through a set of API calls defined by the driver.

The interaction of the real-time driver with non real-time tasks is through a set of three `fifo`s - namely, the read `fifo`, the write `fifo` and the control `fifo`. The control `fifo` is used to exchange control words between the application and the driver. These control words - "change destination address", "write data" and "read data" are used to interpret the data being written on the other two `fifo`s. The applications open these three `fifo`s as files and write and read data with the use of control words using these three `fifo`s.

Non real-time tasks use the non real-time driver with the API as in linux kernel for receiving and sending non real-time packets.

Besides the network card driver, a real-time driver for accessing the joystick resources was also developed.

3.3 Enhanced Techniques for Timing Measurements

In the previous section we showed how some of the periods were chosen in order to meet various timing requirements expected of flight simulators. The pilot-input-to-response needs to be experimentally verified. In this case, we need to measure the elapsed time between time instances, one each in real-time and non real-time space. In RTLinux, times can be measured quite accurately as long as processes lie within the Real-Time space, but not otherwise.

During our measurements, we realized that there is an offset between the time returned by the *gethrtime()* call of RTLinux and the linux system time. If one could measure the value of this offset, then the round trip time as seen by a user (in non-real-time space) could be estimated. We devised a mechanism to measure the offset and used it to take the measurements. The detailed mechanism is explained below since we believe this could be useful in general.

First, we would like to discuss the cause of the offset. In RTLinux, time is measured using the *gethrtime()* call. The time measured is high resolution in *nanoseconds*. On Intel processors (that we used) it measures time from the time-stamp counter. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor. The Intel time-stamp counter is a 64-bit MSR (model specific register) that is incremented every clock cycle. On reset, the time-stamp counter is set to zero.

The time measured in normal linux is the number of jiffies ($10ms$) since system start up, i.e., since the loading of the linux kernel.

Hence, the offset between the two times is equal to the difference in the time since the system came up (the machine was reset), and the time when the kernel was loaded. This will vary with every boot sequence (for e.g., it will depend on time taken at the `lilo` prompt also). Once booted, this offset remains constant.

At anytime after boot up, when the system is running, executing the following steps will give us a measure of the offset.

- Measure the value of the time stamp counter using the Intel *rtdsc* instruction.
- Convert the number of cycles measured as above to a measure of time, say *hwt*.
- Measure the *jiffies* since system startup. *Jiffies* is a global variable in the linux kernel and can be accessed in a kernel module.
- $offset = hwt - jiffies$

In the linux kernel, we implemented the above algorithm as a module, which reads the current time-stamp counter and the number of jiffies at a given time. This information is given back to the client through a device file.

The driver for the above device file is added as a kernel module. A client in user mode, reads the hardware time and the number of jiffies from the device file. This is repeated a number of times and the offset is measured by the client as detailed above.

4. Experimental Results

From what we have said so far, the only requirement that remains to be experimentally verified is the requirement for a bound on the time delay from pilot input to response (to that input). This could be affected by large scheduling or interrupt latencies or execution delays. However, using real-time threads and a real-time network card driver, eliminates these latencies and ensures predictability as shall be corroborated by the results.

We can identify two types of response times given our design. Both start at the Client writer thread polling for the pilot input, but one ends in the real-time space at the Client reader thread, while the other ends in the non real-time space with the A/V module displaying the response. The former round trip time is entirely in the real-time space and we have an analytical bound on its value, namely (Δ + period of the Client Reader thread). But in practice, the data has to reach the A/V Module which is a non real-time module. Therefore, the latter round trip is the relevant measure, but is affected by the unpredictabilty introduced in the system because of the non real-time component. Measurements for both the values are described below.

4.1 Response Time – Reaction to Pilot Inputs

In this set of experiments, we varied the time period of the writer thread and measured the response time (Note: the hardware in the loop is not part of the round trip delay.).

In the interest of space we show the results for $T = 4ms$ case only, even though we varied the period upto $16ms$ and found the results to be similar.

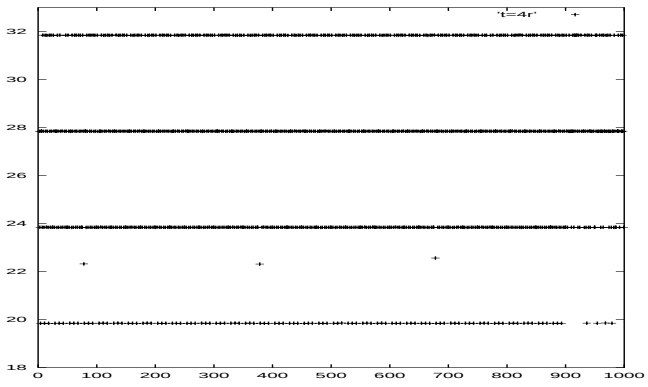

Fig. 6. Response Time(ms) for Different Inputs, Client Writer Period = 4ms

From figure 6, we observe that

- the Response Times fall into groups which are separated by $4ms$, the time period of the reader thread. Since all the threads in the system are periodic, the time instance at which the reader on the client receives the response can be $t + \delta t + 4 * n$, where t is the time instance when pilot input is sent from the writer thread on the client, δt is the offset (within the period) between the reader and writer threads on the client, and n is an integer. Hence, the possible values

of Response times are $\delta t + 4 * n$. Depending on factors like network delays, processing delays on the Server, the value of n will be different, but they will be grouped together in bands separated by $4ms$, since δt is constant, as observed.
• On zooming into each of the bands, we find that the values within each of the bands are scattered within a $10\mu s$ range, validating our assumption that scheduling latencies are of the order of microseconds.
• The maximum value of the Response time is less than $32ms$. This value is less than the analytical bound and meets the requirements of the pilot-input-to-response for the flight simulator system.

The round trip time as calculated in the above cases is the time the writer at the Client gets the joystick input to the time the Server's response is ready for delivery by the reader at the Client. However, this data has to be delivered to the A/V module which runs in the non real-time mode. We next transferred the data received by the Client's Reader to a non real-time A/V process through `fifos` and measured the end-to-end response time. The results are shown in figure 7.

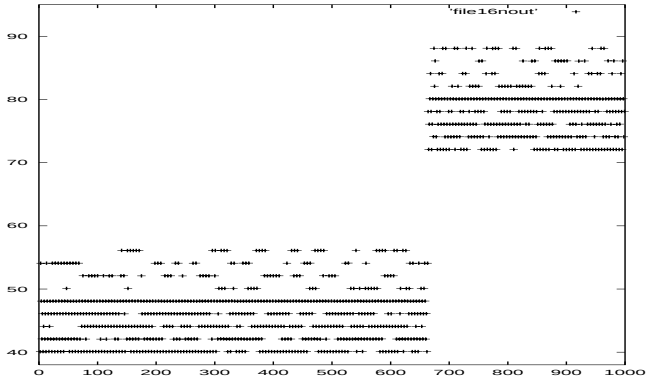

Fig. 7. Response Times (ms) Observed by A/V Module

We notice that while the response time overall is less than the allowable worst case of $150ms$, we have a fair amount of unpredictability when the end-to-end response time is measured at the A/V module end. It should be pointed out that the preceivable "jump" observed close to the 700^{th} observation is an artifact of this specific run. Similar scattered response times were observed for other runs also, but, "jump"s (if any) occurred at different points in time. Nevertheless in all cases, unless a very large non-real-time load was imposed, the end-to-end delays were lower than the required $150ms$. In the present design, making the A/V module more predictable may involve PCI bus locking and interfacing with the graphics card. We are looking into this currently.

5. Conclusion and Future Work

Flight simulators are resource intensive and highly demanding in their timing requirements. We have designed and implemented a distributed simulation system. The system as described can meet the timing requirements imposed on Flight Simulators.

RTLinux acted both as an aid as well as deterrent in the implementation cycle. Instances where it was an aid are:
• The existence of the concept of hard and soft interrupts enabled the synchronisation and hence co-existence of both the real-time and non real-time driver for a single network card. Most of the issues were common to both the types of drivers.
• The real-time driver operates in the real-time space, which is isolated from the normal linux application space. Hence, there was no need to transfer data from the real-time applications to the driver. Also, the `fifo` drivers handled transfer of data from the linux space to real-time space. Handling of data transfer across applications and drivers was separated from application development.
• It provided a familiar POSIX based application development environment. Hence, it made application development easier.

However, there were scenarios in which RTLinux support had to be enhanced to meet our needs:
• There appears to be no mechanism in RTLinux to compare times in real-time and non real-time space. During measurements it was observed that the *gethrtime()* call in RTLinux and the system time of the linux kernel were not the same. An offset between the RTLinux kernel time (the hardware time) and the linux kernel system time was introduced by the boot sequence and hence the times were not the same. We devised a mechanism, details of which were given in Section 3, to measure this offset.
• RTLinux does not support dynamic allocation of memory in real-time space and hence, all the structures in the real-time space are of prefixed size. This was a problem while writing generic drivers for joystick inputs. Hence, one had to hard code the number of buttons and axes of a joystick. Moreover, the buffers used in the network card driver are of fixed size and hence can result in data loss for non real-time packets.
• Giving very small time periods like $2ms$ on slow (e.g., Pentium 150MHz) procesor for tasks, causes the system to freeze. There is no mechanism for the user to come out of such a situation besides rebooting the system.

During the development of the drivers for RTLinux, we learned that the source code of the drivers written for the same device for linux were helpful. The source code of the driver for linux can be very useful, especially for the device dependent parts. For example, in the network card driver, the hardware dependent subroutines like the timers were used from the linux driver code. Also in the joystick driver, there is a vendor specific correction matrix to be applied before arriving at the state of the joystick. This matrix for the particular joystick was available from the joystick drivers for linux.

In the elaborated design and implementation, the only way to transmit interrupt driven inputs from the pilot to the Flight Dynamics system is by sending it from the Client to the Server and then at the server, polling the network card for the input. In such a scenario, the worst case delay in the response to the interrupt input would be dependent on the time period of the reader thread on the Server. In

our case, it would be $4ms$, which is too long a delay for critical inputs. Hence, a mechanism to transmit interrupt driven inputs more effectively has to be devised and implemented. A way of doing this would be to implement blocking I/O in the real-time network card driver for critical data. The processes waiting on blocking inputs would be immediately woken up on arrival of such a packet. The schedulability analysis of such a scheme also needs to be studied.

References

[1] *Real-Time Linux Homepage*. http://www.rtlinux.org.

[2] Manual of Criteria for the Qualification of Flight Simulators. Technical report, CAO: Montreal, 1995.

[3] B.B. Borys and S.S. Dussoye. Flight Simulation Facilities in Europe. Technical report, IFAC Conference on Integrated System Engineering, September 1994.

[4] Mal Gormley. *Aviation Computing Systems*. McGraw Hill, 1997.

[5] Bruce D. Nordwall. *AvWeek: Virtual Reality Pending As Simulators Diversify*. http://www.aviationnow.com/awst.

[6] J.M. Rolfe and K. J. Staples. *Flight Simulation*. Cambridge University Press, 1986.