CMU-ITC-91-107

Distributed Multimedia: How Can the Necessary Data Rates be Supported?

Michael Pasieka (msp@andrew.cmu.edu)
Paul Crumley (pgc@andrew.cmu.edu)
Ann Marks (annm@andrew.cmu.edu)
Ann Infortuna (ai0d@andrew.cmu.edu)
Information Technology Center
Carnegie Mellon University

Abstract

At the Information Technology Center at Carnegie Mellon University[1], we have been developing a system in which it is possible to deliver data to a presentation machine from a remote machine across a public network at a sustained high rate. We have called this system a *Continuous Time Media System* (CTMS). We have found that the UNIX[2] model used for transfer of data between two devices, the network transport protocols, and the ability of adapters to transfer data between themselves are insufficient to do this. We have made modifications to each of these system in order to create a prototype system that we can measure to help define a CTMS protocol. We present the results of this work in this paper.

1. Introduction

The word *multimedia* has been widely used and misused in referring to a computer's capability to present and manipulate non-textual data. A *multimedia system* often implies a system that can manipulate only text and graphic. Such a system may be incapable of displaying both types of information simultaneously. There may be no method for linking the text with the graphic and no means of accessing the information from a remote machine, but the system may still be called multimedia. An ideal multimedia system might include text, graphic and tabular data (including spreadsheets), vector drawings, animations, still images, full motion full color video, Compact Disc quality audio and whole interactive programs (such as a full functional calculator or piano keyboard), all of which could be included within one document. The system might also allow for arbitrarily complex linking of data, and distributed data. Although, few systems exist that approach this ideal, when we discuss multimedia systems we want to consider this whole system.

One key limitation of systems that include full motion, full color video (of NTSC quality) or Compact Disc quality audio is that they are stand alone systems or systems that use private networks to transport the multimedia data. Until now, no one has been able to bring to the market or the lab a real time, public local area networked system for such high data rate media.

We define a *Continuous Time Media System* (CTMS) as a system in which the data to be presented must be received at a reliable, continuous high rate. This presents concurrently the problems of high volume data transport and real time response to the transmission and reception of the data. For example, with Compact Disc audio, the transfer rate is 176.4KBytes/sec (44.1K samples, 16 bits per sample, 2 channels). The source machine must read a disc and redirect the data flow onto the local area network. The destination machine must then receive the data from the network and redirect the flow to the subsystem

[1]This work was performed as a joint project of Carnegie Mellon University and the IBM Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies of Carnegie Mellon University or the IBM Corporation.

[2]*UNIX* is a registered trade mark of AT&T. *RT/PC* and *Token Ring* are registered trade marks of IBM. *AFS* is a trademark of Transarc Corporation.

that is converting the digital data to audio in a such a way that no discernible glitches are heard. This presentation is not simple, even in a stand alone system.

Our group works within the following operational environment. The base operating system is Academic Operating System (AOS) 4.3, a variation of BSD 4.3 UNIX that runs on the IBM RT/PC machines. The system is also an Andrew File System (AFS) client [Morris86]. The local area network is a 4Mbit Token Ring with 70 machines of which several are file servers running AFS. Note that this paper only addresses the problems associated with data transport of CTMS over a Token Ring network in which the source and destination machines are on the same local network (i.e. source and destination are not separated by a router). Although we could have chosen other operating systems and other machines, availability and established expertise were a major factor in these decisions.

Our goal was to support the data rates needed by CTMS over a 4Mbit Token Ring local area network while the ring was in use for other data transport. We ran several tests of the current UNIX system, described above. The initial test was to transport 16KBytes/sec of audio data (8K samples/sec, 12 bit/sample). This worked extremely well within the current UNIX model. We then tested the use of 150KBytes/sec to simulate compressed video or Compact Disc quality audio. This test of data transport failed completely. In this paper, we present our proposed changes, the measurement tools we used to measure the modified system, as well as some of our thoughts about the actual measurement data. With our proposed changes, we created a prototype for successfully transporting CTMS data over a 4Mbit Token Ring local area network, which was loaded with other data.

2. The UNIX Model of Device to Device Transfer and CTMS

In the current UNIX model of data transfer, the only method of transfer between two devices is to create a user level process that reads the data from one device and writes the data to a second device. This leads to having too many data copy operations, which then leads to the CPU's inability to maintain the necessary data rate.

Referring to Figure 2-1, if a user level process reads data from a device, at least two data copies are performed. The first is normally a Direct Memory Access (DMA) transfer between the device and kernel

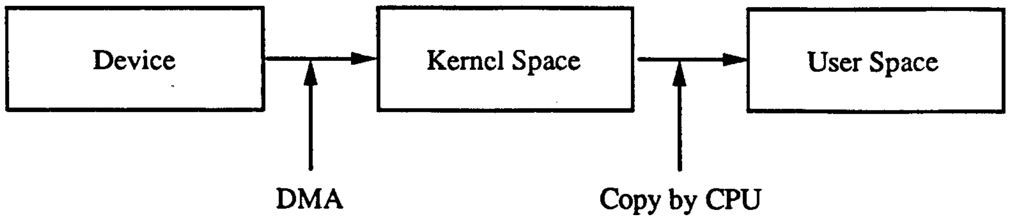

Figure 2-1: Data Copies From Device to User

space[3]. If we can ignore the loading of the system bus while the transfer is in progress, there is no loading of the CPU to do this particular transfer. If the CPU is executing a memory intensive computation at the time, the arbitration between the DMA and the CPU access will degrade the execution speed of both. This DMA transfer always occurs into kernel space.

[3]There is at least one device currently available, the Audio Capture and Playback Adapter produced by IBM, which does not use DMA. In fact, the interface is a byte wide interface. I must assume that the designers of the adapter expected that the audio data would be compressed in software on the adapter before being transferred to the host system. Of course, this requires the device driver builder to also program the DSP (in this case a TI32025) to do this compression. It should be noted that there also exist adapters that use on-card memory mapped into system memory as the method for data transfer.

The second copy is performed by the CPU to transfer the data between kernel space and user space. Unfortunately, the implementation of most device drivers includes a third copy of the data. This third copy is done by the CPU. Referring to Figure 2-2, device drivers normally use fixed DMA buffers in kernel system memory. This forces the device driver to copy the data out of the fixed DMA buffers into a linked list of kernel buffers called *mbufs*.

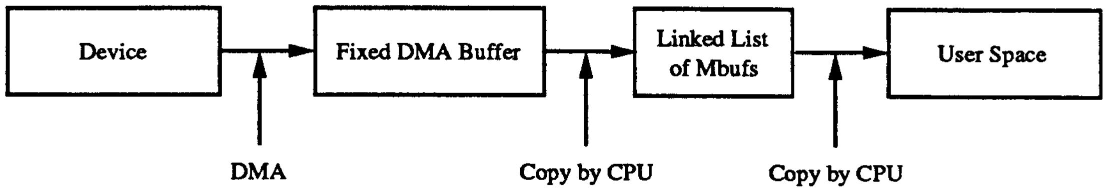

Figure 2-2: Expanded Data Copies From Device to User

The UNIX model uses *mbufs* as a pool of buffers to transfer data between the various layers of protocols. The device driver allocates *mbufs* and copies the packet into these *mbufs*. It should be noted that the allocation of a *mbuf* can be delayed an arbitrarily long time if the pool is exhausted at the time of the request. To transfer the data back to the second device requires three additional data copies. Two of these copies are performed by the CPU and one is performed via DMA.

In total, the number of copies performed to effect the transfer of data between two devices can be as many as six and as few as four. The difference of two copies can be accounted for by the devices' DMA capabilities. There will always be four copies made by the CPU. At a minimum, two of these copies are unnecessary.

We propose a change to the UNIX model of data transfer between two devices. Specifically, direct driver to driver data transfers should be done. This completely eliminates two of the data copies. This change requires that the source device be given a function which when executed will effect the transfer of data between the two devices. In the case where the network is the source of data, an additional function is needed. This function needs an argument of a linked list of *mbufs* that is a newly received packet. This function returns true when the packet should be directly transferred to the device. Handles to these two function calls can be transferred by a user process between the two devices by using newly created *ioctl* calls. We implemented this change of direct driver to driver data transfers in the prototype system for CTMS.

There is one further change that can be made if the model of UNIX data transfers is extended to include transfers by pointer manipulation rather than by data copy. Given that both devices are capable of DMA, all CPU data copies can be eliminated by transferring pointers to DMA buffers between the two devices. If only one of the two devices is capable of DMA, then only one copy can be eliminated.

3. Local Area Network Protocols and CTMS

The currently available standards of Transmission Control Protocol (TCP) and Internet Protocol (IP) as local area network protocols for data transport are insufficient for the data transport of CTMS. Several guarantees are needed from the protocol used to transport the data. These include:

- Bandwidth across the network
- Delivery of a packet within preset time bounds
- Preservation of packet sequence

These guarantees are necessary so that both the amount of buffer space used and the amount of processing time for any single packet are bounded. Of the three guarantees, TCP/IP only provides for one:

the preservation of packet sequence. These protocols can guarantee the preservation only by creating more network traffic in the form of acknowledgments and requests for retransmission of lost packets. The model of the network that was used when these protocols were developed included the idea that the physical network was unreliable. Times have changed.

If a Token Ring network is used, it is possible for the transmitter at a hardware interrupt level to know if the packet was successfully received at the destination[4]. Given this, the levels of software previously used to guarantee packet delivery can be pushed down into the interrupt handler. If the Token Ring device driver is also constrained to send one packet completely before attempting to start sending another, a guarantee of preservation of packet sequence can be accomplished by giving the device driver the packets in the order required.

Additionally, the TCP/IP protocols make the assumption that the network can be dynamically reconfigured. As a consequence of this assumption, IP requests the Token Ring header be recomputed for each packet transmitted. In our case, the transmitter and receiver are always on the same local area network[5]. If we use IP, this would add an additional delay and load on the CPU for no reason, since we know that the route never changes.

Given the requirements that were not met, and other limitations, we decided that TCP/IP would be insufficient as network protocols. We propose that a new protocol be created, CTMS Protocol (CTMSP), and added to the same layer as ARP and IP. This protocol is specifically designed for and limited to the assist of data transfers between the network and other devices. The protocol assumes a static point-to-point connection between two machines. The implementation changes required to add CTMSP to the Token Ring device driver include:

- Adding the ability to specify the priority of the packet sent over the Token Ring. CTMSP uses a Token Ring priority above any other traffic on our Token Ring.
- Adding packet priority within the Token Ring device driver. CTMSP uses a packet priority above both ARP and IP packets.
- Splitting out the function that computes the Token Ring header. This allows for precomputing the header once for the life of the connection.
- Adding code to the split point of ARP and IP packets in order to split out the CTMSP packets and correctly handle them.

We implemented these changes in the prototype system for data transport using CTMSP on a Token Ring network.

4. Adapters and CTMS

Current adapters load the CPU too heavily. There are several sources of this load. If the adapter is capable of DMA and the DMA is done into system memory, this DMA can interfere with the CPU's access to system memory. A second source of loading comes from most adapters' use of interrupts. Once the CPU is interrupted, the amount of delay between the start and end of servicing the interrupt is completely dependent on the implementation of the interrupt handler. If this implementation includes long sections of protected code, the problem becomes more acute.

Critical code segments cause another problem with implementation modifications. Given that we want to have interaction of device drivers at interrupt handler execution time, we must protect the critical sections at the highest possible level. By introducing another source of interrupts, the interactions in the

[4]As long as the destination was on the same physical Token Ring.

[5]If we did not do this then we would have the additional problem of creating a router that could keep up with the data rates that we were using. This is possible but has not been implemented.

UNIX kernel can be nearly impossible to track.

A shortcoming of current Token Ring hardware is its inability to give back an interrupt when a Ring Purge is detected on the network. This leads to the sole source of dropped packets for which no correction can be made. To be able to correct for this, the Token Ring adapter would have to be put in a mode to read all Medium Access Control (MAC) frames that are seen on the ring[6]. This adds yet another loading to the system's ability to respond to interrupts. From observation, the amount of MAC frame traffic on the Token Ring we use is between 0.2% and 1.0%. The MAC frame packets are on the order of 20 bytes of data. Given a 4Mbit Token Ring, there would be between 50 and 250 interrupts to handle MAC frames per second. This additional interrupt and software decoding of packet headers would add an unacceptable amount of overhead to detect the small number of Ring Purges that occur on the Token Ring. Other than this single source of dropped packets, we found that the Token Ring was completely reliable for sending packets between two machines on the same ring.

On IBM RT/PC machines, we can make a change to reduce the loading on the CPU. That is to modify the Token Ring device driver so that it does not use system memory for the fixed DMA buffers. This change is specific to the architecture of the IBM RT/PC. This architecture has two separate buses that transfer address and data information within the machine. The first is between the CPU and the main system memory. The second, normally called the Input/Output (IO) Channel Bus, interconnects all of the attached adapters. The arbiter for accesses between these two bus structures is the Input/Output Channel Controller (IOCC). An adapter is available that is solely memory, called IO Channel Memory. DMA between another adapter and IO Channel Memory can occur on the IO Channel Bus and not cause interference with accesses by the CPU to main system memory. We implemented this change to the Token Ring device driver to use IO Channel Memory in order to reduce the loading on the CPU in the prototype system for data transport using CTMS.

If adapters were designed and manufactured to do data transfers using DMA or directly transfer data between two devices, the load on CPU could be reduced further.

5. Measurement Tools

Once we built the prototype system, we needed tools to measure both the packet activity on the network, as well as several points with the kernel. Commercial products exist that measure the load on a Token Ring and capture packets for later examination quite well. After examining several, we used IBM's Trace and Analysis Program (TAP). This tool allowed for the recording and time stamping of all packets seen on the network, including all MAC frames. The tool also recorded the first Token Ring adapter's buffer of actual packet data (up to 96 bytes) as well as the Token Ring's Access Control byte, Frame Control byte and total length. However, there are limitations of the tool's ability to record all packets[7].

Using the TAP tool, we were able to detect when packets were out of order and lost. In the first case, out of order packets were a direct result of the Token Ring device driver implementation. Once the critical sections of code were more carefully protected, the problem of out of order packets completely disappeared. Thus, we found that we were working with a network that would transmit packets in order and do so reliably with only one exception: Ring Purge on the Token Ring.

Ring Purges occur on the network primarily due to new stations inserting into the network or old stations reinserting into the network. If a packet is being transmitted at the time of insertion, it is possible that the packet will be lost. Ring Purge is transmitted by the Token Ring's Active Monitor after such an

[6]This completely ignores the fact that to get a Token Ring adapter that would pass up the MAC frames requires proprietary software in the ROMs on the adapter.

[7]For specific details, please see the documentation of the product [IBM90].

event has occurred[8].

If the adapter is programmed to interrupt the transmitter when a Ring Purge is seen on the ring, then the transmitter can attempt to correct for a possible lost packet by retransmitting the last packet that is still in the fixed DMA buffer. The receiver, in this case, might need to ignore a duplicate packet if the transmitter incorrectly retransmits a packet. Unfortunately, the adapter does not interrupt the processor with the information that a Ring Purge has occurred. As was mentioned earlier, the only way to detect this occurrence would be to set the adapter to receive all MAC frames and pass them on to the interrupt handler. But, the software on the adapter does not allow for passing MAC frames to the host processor. Even if the software did allow for this, the overhead in handling interrupts and parsing MAC frames to detect a Ring Purge would be unacceptable. We decided to allow for the loss of a single packet and to measure the frequency of this occurrence. Measurement revealed that Ring Purges are normally a result of a station insertion, and occur on the order of 20 times a day. We decided that we could safely ignore this level of lost packets by adding code to recover.

Even though the TAP tool was very useful for the macro scale measurements of the Token Ring, it was not sufficient to measure the actual device driver. We were interested in gathering events and time stamping them with 100 microsecond accuracy. We needed to be able to correlate the time stamps of several other points within the device driver, on possibly more than one machine simultaneously, to determine the correct transmission and reception of packets. These types of micro scale measurements were needed to find the worst case delay of packets as well as the mean and standard deviation of inter-packet departure and arrival times. The available commercial products fell far short of being able to do all of this. The most critical part missing was a time stamping facility with the accuracy required.

5.1. Source of CTMS Data

At this point, we need a short discussion of the source of data for the following measurements. What we required was a stable source of interrupts. We used IBM's Voice Communications Adapter (VCA). Briefly, the adapter has a TI32010 DSP, 2k by 16 bit memory, which is byte accessible by the host processor, can be interrupted by the host and can interrupt the host. We created a program to run on the adapter that would interrupt the host every 12 milliseconds. We added several *ioctl* calls to set up the device in this special mode, to request the Token Ring header and keep this header as part of the state of the device, and to request handles to functions needed by the modified Token Ring device driver. We hard coded in the VCA's device driver calls to the Token Ring device driver for calculation of the Token Ring header and for the sending of a packet.

We modified the VCA's interrupt handler to create a CTMSP packet by allocating a linked list of *mbufs* for the packet and then copying the static precomputed Token Ring header, a destination device number, and a packet number into the packet. We then appended the packet with data to create a packet of 2000 bytes in length (including the header information but excluding the Token Ring protocol bytes). We then sent this packet via the modified Token Ring device driver. The result of this modification was to create a CTMSP data transport stream of approximately 150KBytes/sec.

5.2. Tools Used and Built to Measure the Modified System

After discovering the lack of measurement tools that existed to measure device driver performance on multiple machines at the same time, we decided to look into building our own tools. We used several methods to measure both the VCA and the Token Ring device drivers. We will discuss the error

[8]For specific details on this, please see [IBM89].

introduced by the various methods of measurement within the description of each method. The points of measurement within the total system were as follows:

- The Interrupt Request Line (IRQ) of the VCA adapter
- Entry into the VCA's interrupt handler
- Immediately after the packet is copied into the fixed DMA buffer and immediately before the Token Ring adapter is given the *transmit* command.
- Immediately after the received packet is determined to be a CTMSP packet.

Given that we wanted to measure these points, what we needed was to add extremely small pieces of code in the appropriate places that would cause the events to be time stamped and recorded.

5.2.1. IBM RT/PC Used as Measurement Tool

We made the first attempt at time stamping events by using a pseudo device driver. We modified the Token Ring device driver to call a procedure within this pseudo device driver. A UNIX *open* call to this device set a flag in the Token Ring device driver that enabled these event time stamping procedure calls. We added an *ioctl* call within the pseudo device to read out the recorded data. We could only measure the last three points mentioned above since this was all done in software.

We were able to coordinate the activities of the transmitter, receiver and the TAP tool under a centralized control point. The end result was a set of computers that recorded and analyzed data in real time. If a packet was lost, had an extremely long inter-departure or inter-arrival time, or there was an incorrect ordering of packets on the transmitter and/or receiver, all machines were halted and a snapshot of the data was taken. We then examined the snapshots to decide what error had occurred. Histograms as well as means and standard deviations were computed for the inter-packet departure and arrival times from this data.

The main problem with recording data using this method was the potential for interaction between the data recording mechanism and the rest of the activity of the machines. The error introduced in the time stamps due to this method was a direct result of this interaction. If the time stamping procedure was done with interrupts disabled, then the procedure itself might delay unacceptably another point of measurement. If interrupts were enabled during the procedure, then the time stamp could be significantly in error due to the possibility that another interrupt occurred while executing the recording procedure. In addition, the clock granularity was only 122 microseconds. All in all, this was a poor method of recording data on inter-packet arrival and departure times, but was extremely good at helping to find bugs in the Token Ring device driver, bugs in the system as a whole, as well as debugging our modifications to the system.

5.2.2. Logic Analyzer and Oscilloscope as Measurement Tools

The use of a logic analyzer is the least obtrusive way of measuring the values of interest, since we could program the analyzer to trigger on any memory read or write access or on any edge of any signal. We used this to determine the degree of accuracy of the VCA interrupt source. The result was that the VCA could interrupt the host processor every 12 milliseconds as desired with no detectable variation. We made further measurements using an oscilloscope to look at the second pulse of the Interrupt Request (IRQ) line given that we were triggering on the leading edge of the first pulse. The second pulse varied on the order of 500 nanoseconds from 12 milliseconds. We considered this conclusive proof that the VCA interrupt source was completely solid and that we need not consider that there was any error introduced by the source of interrupts.

The second use of the logic analyzer was to measure the variation between the IRQ pulse and the start of the VCA interrupt handler. Even while loading the Token Ring and the local disk, the largest variation seen was 440 microseconds.

We discarded using the logic analyzer to measure any of the other parameters due to the limitations of the device and our own expertise. Specifically, we needed a complete histogram of all of the intervals described above so that the total shape of the histogram curves could be examined. The logic analyzer was not capable of this functionality.

5.2.3. IBM PC/AT Used as Measurement Tool

Figure 5-1 shows how we delegated the time stamping of events to a pair of external machines. We used an IBM PC/AT machine with a parallel interface board consisting of eight separate 8-bit wide interfaces. We installed a serial/parallel interface board in each machine on which we wanted to time stamp events. Within the Token Ring device driver, we replaced the calls to the pseudo device driver procedure with in-line code to write specific values into the parallel port and toggle the strobe output line. In the case of transmission or reception of packets, we devised the shortest possible test to determine if the packet was an CTMSP packet. If so, the last 7 bits of the packet number were written to the parallel port and then the strobe output line was toggled.

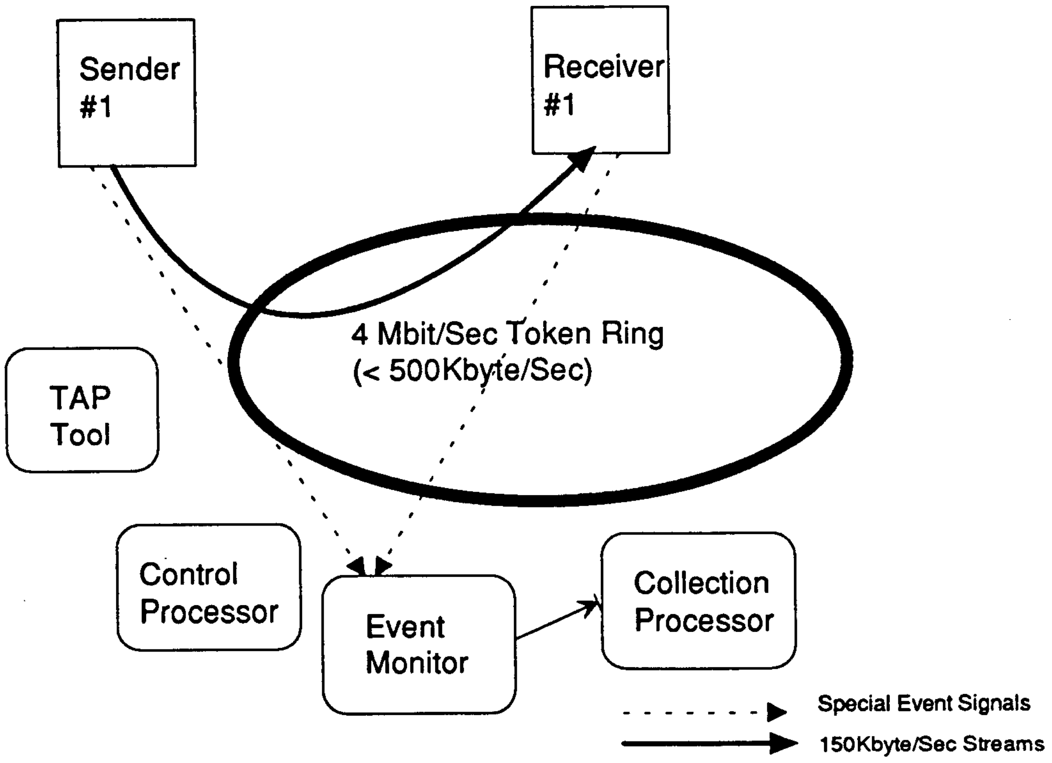

Figure 5-1: Machine Interconnections While Using PC as Measurement Tool

We then collected the data on the PC/AT within an interrupt handler infinite loop. This loop time stamped the data with a 16 bit clock value where the resolution of the clock was two microseconds. We used another timer within the PC/AT to generate a signal with a period of 50 Hz. We then tied this signal to the eighth parallel input port. This guaranteed that the programs that later analyzed the data could correctly detect the roll over of the 16 bit clock time intervals, even if no other datum occurred between two roll over periods of the clock.

Within the interrupt handler infinite loop, we polled the register that held the bits indicating which channels had an interrupt pending. If any bit was set, we read the clock as well as those ports that had interrupt bits set (excluding the eighth port). We then queued the value of the interrupt register, along with 16 bits of clock time and the values of the ports which were read. If no bits were set in the read of the interrupt register and if there was any queued data, the data was sent out a separate parallel port to a second PC/AT machine. This output to the second machine was fully handshaked. We saved the data received by the second PC/AT over this parallel connection onto a local disk of the second machine.

To determine the error introduced by this method of measurement, we measured the worst case time for the execution of the interrupt handler loop. We found this number by pulling out the VCA's Interrupt Request Line and time stamping occurrences of these events. By using a logic analyzer, we determined that the VCA was able to dependably interrupt on 12 millisecond boundaries within negligible error. Therefore, any variation was due to the measurement tool. Upon examining the spread of the histogram curve generated by this test, we found that there was a 120 microsecond spread on both sides of the 12 millisecond mean. We conducted a second test by using the logic analyzer to measure the loop execution time in the best and worst case conditions. The results were that the interrupt handler loop had a 60 microsecond worst case execution time. Considering both sets of measurements, we concluded that the tool had acceptable error to measure the rest of the system.

5.3. Measurements of the Modified System

Given all of the errors, we wondered at times if we would be able to say anything about what we were doing beyond the obvious; "Sending this amount of data is hard and one must be very careful." We decided that we could.

In all cases, we used a point-to-point static network connection between the transmitter and receiver. Beyond the measurements discussed earlier, the following differences will alter the results:

- Transmitter uses IO Channel Memory vs. System Memory for fixed DMA buffers
- Transmitter copies only header into fixed DMA buffer vs. copying both header and data
- Transmitter copies data from the VCA device buffer to *mbufs* vs. direct copy of data from the VCA device buffer to fixed DMA buffers
- Receiver copies header and data from a fixed DMA buffer into *mbufs* before passing to the VCA device vs. VCA examining the packet while still in a fixed DMA buffer
- Receiver copies data out of *mbufs* into the VCA device buffer vs. no copy of the data (dropping the packet)
- Use of priority within the Token Ring device driver vs. use of same level of priority as all other packets being sent by the local machine
- Use of priority on the Token Ring vs. use of the same level of priority as all other packets on the ring
- Method of measurement: Local (RT/PC), remote (PC/AT), monitoring of network (TAP), logic analyzer
- Private vs. Public Network
- Level of background load on network
- Transmitter/Receiver in stand alone vs. multiprocessing modes

We selected the following scenarios for the presentation of the data in this paper:

Test Case A) Transmitter uses IO Channel Memory for fixed DMA buffers; transmitter copies both the header and data into fixed DMA buffers; transmitter does not copy data from VCA device into *mbufs*; receiver copies data from fixed DMA buffer into *mbufs*; receiver does not copy data from *mbufs* into the VCA device buffer; priority within the Token Ring device driver of CTMSP packets; Token Ring priority; remote measurement; private network; no loading of network; transmitter and receiver in stand alone mode.

Test Case B) Transmitter uses IO Channel Memory for fixed DMA buffers; full copying of data on Transmitter and Receiver; priority within the Token Ring device driver of CTMSP packets; Token Ring priority; remote measurement; public network; normal loading of network; transmitter and receiver in multiprocessing mode but not heavily loaded.

In both cases, we examined histograms of the following measurements:

1) The inter-occurrence of pulses of the VCA's Interrupt Request Line

2) The inter-occurrence of the entry into the VCA's interrupt handler

3) The inter-occurrence of the point immediately after the packet is copied into the fixed DMA buffer and immediately before the Token Ring adapter is given the *transmit* command

4) The inter-occurrence of the point immediately after the received packet is determined to be a CTMSP packet

5) The differences between like occurrences of (1) and (2)

6) The differences between like occurrences of (2) and (3)

7) The differences between like occurrences of (3) and (4)

Test cases A and B, histograms 1 through 5 as well as test case A, histogram 6 all showed values which could easily be explained given the total system and its interactions. Test case B, histogram 6 is shown in Figure 5-2. This particular histogram is interesting because of the bi-model curve. The explanation for this is simply that there is interaction between the transmission of CTMSP packets and the transmission of other system packets. The other traffic includes AFS *keep alive* packets, ARP traffic and socket *keep alive* packets. The socket packet traffic is an artifact of the test set up. All of the machines in the test are being directed by a central control machine. The communications link between the control machine and each of the other machines in the test is via UNIX sockets. All of this system traffic causes some of the CTMSP packets to be queued rather than sent immediately. The system then plays catch up for tens of CTMSP packets. There are 68% of the data points within 500 microseconds of 2600 microseconds, 15% within 500 microseconds of 9400 microseconds, 16.5% between 2800 and 9300 microseconds with the remaining 2% in the tails of the graph which extend from 100 microseconds to 14000 microseconds. The 2600 microsecond mean of the first peak in the histogram gives the mean latency of sending a 2000 byte packet. The transfer rate of copying data from the system memory where the *mbufs* are located to the IO Channel Memory, where the fixed DMA buffers are located, is on the order of 1 microsecond per byte. This leads to 2000 microseconds of latency specifically attributable to copying the packet. The additional 600 microseconds can be attributed to the execution of the code between the two points of measurement.

Test case A, histogram 7 is shown in Figure 5-3. This graph shows that the minimum latency of a 2000 byte packet is 10740 microseconds. Specifically, 98% of the data points fall within 160 microseconds of the 10894 microsecond mean with the remaining 2% spread to the right of the mean extending to 14600 microseconds. The latency seen can be attributed to DMA'ing the data on both ends, transmission time across the Token Ring, delay of the receiver's interrupt handler being entered and execution of the receiver's interrupt handler code. The spread of the curve can be attributed to other interrupt sources and the execution of protected code segments throughout the kernel. Additionally, there is delay if a packet is on the Token Ring at the time of transmission. The only type of packet other than the CTMSP packets in this test case is the normal MAC frame traffic, which uses 0.2% of the network in this completely unloaded test case.

Test case B, histogram 7 is shown in Figure 5-4. This graph shows that the minimum latency of a 2000 byte packet is 10750 microseconds. Specifically, 76% of the data points fall within 160

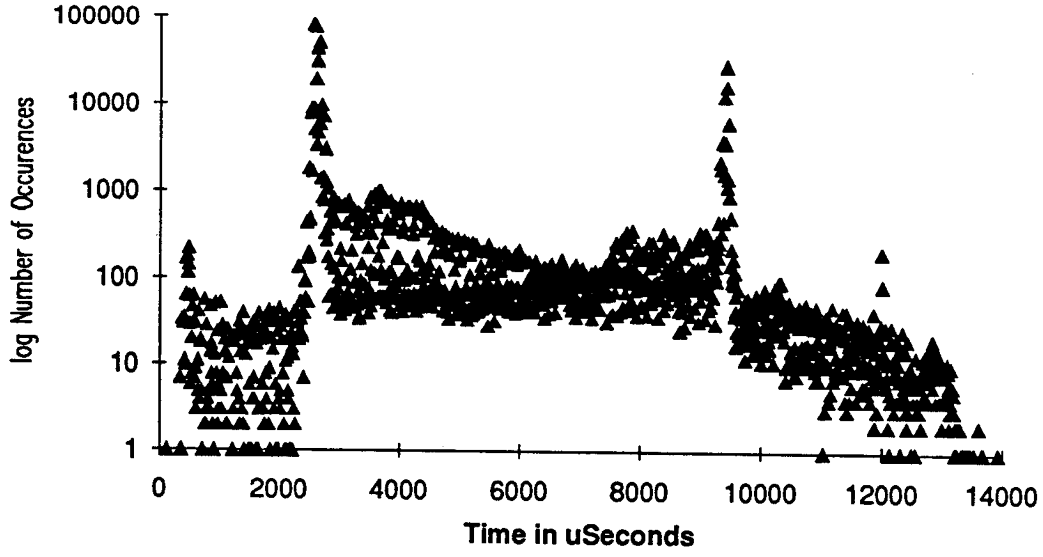

Figure 5-2: VCA Interrupt Handler Entered to Just Prior to Transmission

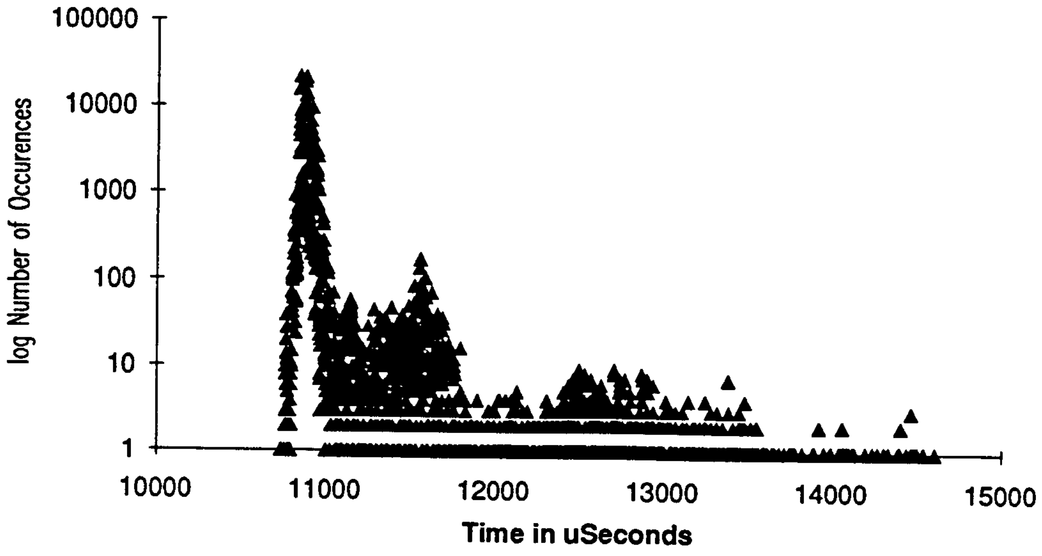

Figure 5-3: Transmitter to Receiver Times, Test Case A

microseconds of the 10900 microsecond peak, 21.5% fall within the range from 11060 to 15000 microseconds, 2.49% fall within the range from 15000 to 40050 microseconds, with the remaining two points falling in the range from 120 to 130 milliseconds. These last two exceptional points are not in the histogram as shown. Examining the normal traffic on the network shows that there are three different sizes of packets. The first size consists of MAC frame packets of approximately 20 bytes in total length. The second size consist of ARP packets and those packets which comprise the *keep alive* packets for AFS. Each of this second class of packets are approximately 60 to 300 bytes in total length. Lastly, there are the file transfer packets sent while a compile is done or during UNIX kernel copying activity. These packets are 1522 bytes in total length. The traffic associated with these three sets of packets can justify some of the spread of the curve as well as some of the secondary peaks. Most of the remaining points can be attributed to the interaction of the reception and transmission of packets other than the CTMSP packets. The two

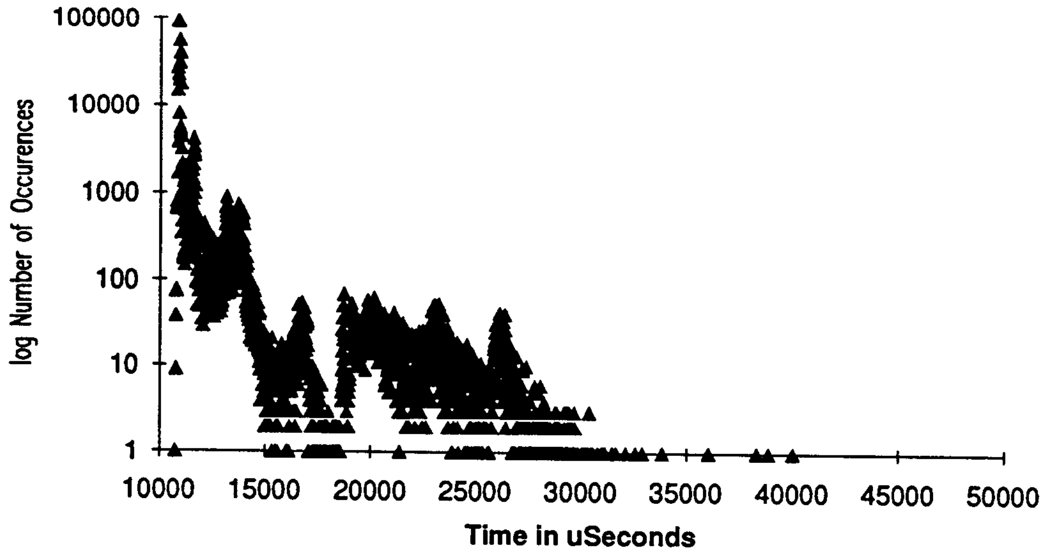

Figure 5-4: Transmitter to Receiver Times, Test Case B

exceptional points in the range of 120 to 130 milliseconds can not be completely accounted for. We can only speculate that a large number of circumstances occurred at the same instance. The majority of this time can be attributed to both a soft error on the Token Ring and the Token Ring timing out and resetting of the network. Unfortunately, this can only account for 10 milliseconds. If we speculate that a ring insertion would cause multiple Ring Purges, then we could completely account for this activity. Experimentally, we have seen on the order of 10 Ring Purges back to back. We conclude that this is precisely what is occurring. We then measured the number of insertions seen in one day. The number was under 20, approximately one an hour. The test data shown for Test Case B was run for 117 minutes. We conclude that the two exceptional data points are two insertions into the Token Ring.

6. Conclusions

In order to transport 150KBytes/sec of CTMS data, two modifications are necessary. The UNIX model of device to device data transfers must be changed to eliminate a minimum of two data copies. This can be done by transferring the data directly between two devices rather than indirectly via a user process. Secondly, a new network protocol must be used. It should be noted that the intent of this work was not to define the architecture of this new protocol but rather to build a prototype system that could be measured to help with the later definition of the protocol. Finally, a third modification is useful. This third modification is the use of IO Channel Memory for the fixed DMA buffers.

As for conclusions on the measurement of the prototype system, the worst case times between transmission and reception of a single packet is 40 milliseconds. There are two exceptional data points within the 120 to 130 millisecond range. Both of these points are explained by the Token Ring timing out and resetting itself during a ring insertion or reinsertion by a station. Even with these exceptional data points, the buffer space needed for 150KBytes/sec CTMSP data transfer is under 25KBytes. This amount of buffer space is well within a reasonable range to support the functionality of data transport of Continuous Time Media Systems.

References

Comer88. D. E. Comer, Internetworking with TCP/IP: Principles, Protocols, and Architecture, Prentice Hall, Englewood Cliffs, NJ, 1988.

IBM89. IBM Token Ring Network Architecture Reference, SC30-3374-02, third edition 1989.

IBM90. IBM Token Ring Network 16/4 Trace and Performance Program User's Guide, 93X5688, first edition June 1990.

Leffer89. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, The Design and Implementation of the 4.3BSD UNIX Opertating System, Addison-Wesley Publishing Company, 1989

Morris86. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, F. Donelson Smith, Andrew: A Distributed Personal Computing Environment, Communications of the ACM, Volume 29, Number 3, March 1986.

Stevens90. W. Richard Stevens, UNIX Network Programming, Prentice Hall, 1990.

Tevanian87A. Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, Richard Sanzi, A Unix Interface for Shared Memory and Memory Mapped Files Under Mach, Carnegie Mellon University, Department of Computer Science, July 1987.

Trevanian87B. Avadis Tevanian, Jr., Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach, Carnegie Mellon University, Department of Computer Science, CMU-CS-88-106, December 1987.

TI88. First-Generation TMS320 User's Guide, Technical Report, Texas Instrutments.

Biographies

Michael Pasieka received his BS degree in Computer Science and MS degree in Electrical Engineering and Computer Science from MIT in June, 1984. He has worked for Texas Instruments, Amdahl, Symbolics and is currently working for the Information Technology Center at Carnegie Mellon University. He has specialized in inter-computer communications, working on such projects as serial links, input/output testing for Amdahl class machines, and currently transport of data at a sustained high rate in support of distributed multimedia.

Paul G. Crumley has worked in the field of computing systems for about 15 years. Though he has an Electrical Engineering degree from Carnegie Mellon University (BS EE 1983) he can appreciate a well designed and implemented piece of software. Paul has been a member of the Information Technology Center at CMU for seven years where he currently leads the Continuous Time Media System group. Paul is interested in architectures for parallel processing of data, the implementation of distributed systems and photography.

Ann Marks received her BS in Electrical Engineering in May, 1976, Masters of Engineering (Electrical) in May, 1977, and Ph. D. Electrical Engineering/Computer Science in August, 1980, from Cornell University. She is a member of IEEE and ACM and has worked at the Information Technology Center at Carnegie Mellon University since July 1987. Her current research interests are the transport of continuous time media and server design for continuous time media. Prior work includes document interchange using ODA for the Expres Project. She is a co-author of a book on document interchange entitled *Mulit-media Document Translation ODA and the Expres Project* (Springer-Verlag, 1991).

Ann Infortuna received a BS degree in Computer Engineering from Lehigh University, in 1984, and an MS degree in Electrical Engineering from the University of Rochester in 1988. She previously worked in system design and development of new products for the Xerox Corp., and is currently working for the Information Technology Center at Carnegie Mellon University where she is invovled with the design of an execution environment for distributed multimedia.