Message-Bundle Converting in Intenet Protocol Multicast-Based High Level Architecture Exercises

Tracy A. Bobo

AFIT/GE/ENG/97D-20

Message-Bundle Converting In Internet Protocol Multicast-Based High Level Architecture Exercises

THESIS

Tracy A. Bobo, Captain, USAF

AFIT/GCS/ENG/97D-20

AFIT/GE/ENG/97D-20

Message-Bundle Converting

In

Internet Protocol Multicast-Based

High Level Architecture Exercises

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air Education and Training Command

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Electrical Engineering)

Tracy A. Bobo, B.E.

Captain, USAF

December, 1997

AFIT/GE/ENG/97D-20

Message-Bundle Converting In Internet Protocol Multicast-Based High Level Architecture Exercises

THESIS

Tracy A. Bobo, Captain, USAF

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air Education and Training Command

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Electrical Engineering)

[illegible]
Martin R. Stytz, LtCol, USAF
Member

[illegible]
Keith A. Shomper, Major, USAF
Member

[illegible]
Richard A. Raines, Major, USAF
Chairman

Acknowledgments

My first acknowledgment is to God, who has given me the talent, opportunity and friends to pursue the joys in life I have so richly cherished. My wife deserves a very heartfelt thanks for her patience, sacrifice and support throughout my service at AFIT and my Air Force career. While at AFIT I have been surrounded by some of the best minds the Air Force has to offer. My thesis advisor, coach, sounding board, and motivation expert, Major Rick Raines, has been and continues to be a tremendous help. His insight guided me through many blurry corridors and shed much needed light in some very gloomy academic pits of despair. While I was anxious to slay dragons, Major Keith Shomper provided the wisdom to keep my enthusiasm sharply focused on productive areas of research. Without his knowledge of the simulations arena, most of the publications I researched would have been nothing more than Greek poetry to me.

I also owe a deep thanks to the true code wizards; Jeff Bush, John Lewis, Scott Rothermel and Troy Jonson. These guys reviewed and provided a lot of the syntax in the program code. They also patiently allowed me to 'experiment' on their production simulation system using the graphics network.

Table of Contents

List of Figures

List of Tables

Abstract

The Department of Defense is pushing for more widespread and realistic interactive training simulations, which increases the demand on network capacity and resources. While network bandwidth is a measurable resource, packet bandwidth, or the number of packets-per-second (Pk/s) a host can handle, is a shifting commodity. This research analyzes host performance characteristics under varying data loads. The hosts include SGI single and multi-processor systems and Intel Pentium platforms using both Windows 95 and Linux Operating Systems. The networking media covers Ethernet, ATM and FDDI. For the ATM network, both AAL5 and IP over ATM were analyzed. With the data from this research, a system is proposed and developed that takes individual messages and bundles them into multi-message packets. This bundling process overcomes the 5,000 Pk/s limitation, reduces the CPU network handling time and introduces a flow-control mechanism at the local network level.

While the idea of bundling messages to increase CPU efficiency is not new, there are no current methods of bundling within the new High Level Architecture (HLA). This proposed process is a novel approach to introduce flow control, priority message handling and increase address space while utilizing bundled data delivery. For traditional network delivery, typical CPU usage from network data varies as a function of traffic load, ranging from 5% at 500 messages-per- second to over 80% at 4,000 messages-per-second. The new bundling process requires 10% at 500 messages-per-second but only increases to 13% at 4,000 messages-per-second.

1. Introduction

The United States Armed Forces are modernizing the approach to training and exercising the military services by using interactive battlefield simulations. Leveraging current networking technology and pushing the limits of both hardware and software have provided realistic and detailed interactive computer combat simulations. While the detail and fidelity of the simulation places a great demand on the computer hardware alone, the interactive nature of the application stresses both the computer host and the underlying network. Technology has always been one small step ahead of implementation in this mad race for modernization. New off-the-shelf components can always process a little more, go a little faster. Recent changes within the simulation community are poised to take advantage of these latest and greatest technological gains, but some disturbing issues involving network performance are on the horizon.

1.1 Simulation system goals

The Defense Interactive Simulation (DIS) system has been the preferred large scale interactive distributed simulation system for the past few years, but this system is rather rigid and does not scale very well [11]. To get a scaleable system, the High Level Architecture (HLA) has been developed. One change within the HLA system includes a new definition of data descriptors for the messages that flow between simulations. These new descriptors provide for a flexible method of identifying a simulation component and its attributes. Another change within HLA is the use of multi-cast traffic rather than the broadcast method typically used within DIS. These changes allow for a scaleable approach to simulation design. The simulation community is pushing this scaleable concept as far as possible. There are plans for 50,000 simulation entities by 1998 and 100,000 entities by the year 2000 [2].

1.2 HLA Distribution Concepts

The HLA message distribution system is based upon the Internet Protocol's (IP) multicast addressing scheme [12]. This allows the network to provide some filtering based upon the host's interest. This is an improvement over the broadcast mechanism DIS uses since each host will not be interrupted for every message on the network. The routers and networking components provide for management of the multicast address space, but timely access to the data by the simulation is critical [17]. While local IP multicast address management is very quick, wide area IP multicast address management may introduce data latency [4]. IP multicast data transfer is based upon the User Datagram Protocol (UDP). UDP provides point-to-point and point-to-multi-point communications within a network, but the sender has no feedback from the receiver.

1.3 HLA Limitations

Multicast distribution exposes some limitations inherent in the networking infrastructure. The first limitation is address space management. Since routing tables must traverse the same network as the data, there is a cost associated with having extensive address spaces. As the number of HLA entities increase, more network bandwidth and more address space will be required. Providing more address space comes at the expense of bandwidth, so these are opposing desires. Simulations also require timely access to network data, but as the network scales in size, address management will introduce more propagation delays as the host changes its address table. Another limitation of using multicast addresses is the issue of bundling. A source will not know which host is interested in the data, so each message must be sent as a distinct unit. While this is more efficient in terms of network bandwidth, it places an increased load at the host in terms of packets-per-second (Pk/s). Bandwidth is an easy resource to identify since it is one of the networks defining elements. Host Pk/s processing capability is very hard to define, since it is a function of the host, operating system, network interface type and even what type of data is currently on the network. For this reason, it is often assumed that if the network can simply deliver the data to the host, then the host can process it. While this assumption is true in most cases, there are limitations where the host simply can not handle all the data that a network can provide [12]. Furthermore, if the host can process the data, it does so at a cost in terms of CPU usage. Since the simulation also demands CPU resources to keep the fidelity of the interaction believable, we face a dilemma. Too much data slows the simulation beyond believable tolerance; too little data and we lose significant meaning in the environment.

1.4 Problem Statement

This thesis analyzes specific limitations inherent in the current networking environment. These limitations include Pk/s processing capabilities of various hosts, operating systems, networking media / protocols, and IP multicast processing. In addition, a system is proposed and developed to move CPU intensive message handling off the simulation hosts and place the burden on an inexpensive set of computers.

1.5 Thesis Presentation

This thesis is divided into five chapters with one appendix. Chapter 2 introduces ATM and other networking infrastructures as well as the operations of the IP multicast protocol. Chapter 3 discusses the approach to measure the limitations of IP multicast, and host Pk/s processing. Chapter 3 also introduces the design concept of the 'converter' process that will shift the message handling tasks onto inexpensive PC type hosts. Chapter 4 presents the results of the testing; including the comparison of several simulated converter based networks to their baseline network counterparts. Chapter 5 includes a final analysis with recommended implementation strategies. Included in Chapter 5 is a list of future areas of research for the converter-based network. Appendix A details specific implementation data and Appendix B lists the source code used in the converter process.

2. Background

With the advent of Asynchronous Transfer Mode (ATM) there are now many ways to use networking resources in a particular distributed processing environment. The High Level Architecture (HLA) and the Defense Interactive Simulation (DIS) are examples of a distributed environment that relies very heavily on the underlying network. In order to evaluate optimization methods for the DIS and HLA network, an understanding of current networking media protocol (Ethernet) and a possible alternative media protocol (ATM) is necessary.

2.1 Asynchronous Transfer Mode (ATM)

2.1.1 Introduction

ATM is a networking strategy based on the concept of media level connection-oriented delivery. A connection-based delivery system means that the path between sender and receiver must exist. In a connection-less environment, the sender has a 'fire-and-forget' strategy; the host packages the data and then transmits it directly to the network for delivery. The current networking paradigm based on this connection-less data delivery is the Internet Protocol (IP). The difference between connection and connection-less delivery can be thought of as the difference between making a phone call and mailing a letter. A phone call requires both parties to have a communication path (logical or physical circuit) open between them, a person drops a letter in the mailbox and the mail system handles the delivery.

The debate over which media scheme is best, connection or connection-less, has never reached a definite conclusion. For a small community of highly interactive computer systems, a connection-based network has the lowest packet delays and therefore performs best [7]. This is evident by the recent surge of switching devices for local networks. A switch provides a temporary dedicated path between end users to increase the overall throughput of the system. In a common access network, like Ethernet, each computer must compete with other computers for the limited amount of shared communications bandwidth that is available. Studies have shown that as the utilization rates rise for the Ethernet network, packet delays go up quickly [7]. At a usage rate of 70%, the data has almost totally saturated an Ethernet network. Packet delay rates go up exponentially with only a small increase in traffic beyond this point. A switched system however, can be pushed to over 90% utilization with only a linear increase in packet delays [7].

2.1.2 Optimizing with ATM

A switched network, however, does have a drawback. Dedicated bandwidth is a wasted resource if unused. A long-haul T-1 (1.544Mbps) line is a very expensive resource and users typically want as much utilization over a leased communications line as possible. The classical answer has been to multiplex several low data rate users onto one faster rate line. Each user has the benefit of the increased data rate while the overall utilization of the line is high. As long as all the users do not attempt to use the entire bandwidth at once (low to medium utilization), this arrangement works fairly well [15]. ATM is an attempt to give networks the benefit of multiplexed channels (dedicated bandwidth) and the flexibility of global connectivity.

2.1.3 ATM Connections

The ATM network establishes a virtual connection between sender and receiver to ensure low data latency while combining multiple users' data across one physical communications line to ensure high utilization. This 'best of both worlds' approach by the basic ATM network is usually too simple to handle intricate networking scenarios [6]. ATM signaling, the process of setting up and tearing down the Virtual Connection (VC) has evolved very quickly to fill in these gaps. One important note is that ATM can carry IP traffic. This occurs because the true ATM protocol resides at a lower layer in the networking model than IP. A person can think of ATM as either the interconnection of wires and circuits providing a path for higher level protocols, or as a native data delivery mechanism. All further references to ATM in this document will mean native ATM, unless otherwise specified.

2.1.4 Network Management

The management of the underlying network is typically the job of the next higher layer in the protocol stack. The most influential layer in network management is the transport layer. Currently, the most pervasive transport protocol is the Transmission Control Protocol (TCP). Networks usually couple TCP with IP to form TCP/IP, an addressing and data delivery platform that has made the World Wide Web, or Internet, what it is today. Looking closer at how TCP uses IP to do its job will shed some insight into why the distinction between connection versus connection-less data delivery is important. A typical TCP/IP application is the web browser. A user enters a requested web site and the network transparently responds with the appropriate data. A quick look at what actually happens in this request will demonstrate how the network 'works'.

2.1.4.1 IP Address

The user enters a destination name; a name server then converts this name to an IP address. Once the system knows this address it sends a request packet to the remote computer via a process called routing. Routing is based on the IP address, which has two parts: the network id and a host id. In decimal, or 'dot', notation this is something like 157.221.143.15. In this example the first two numbers are the network id. This network id is like the zip code in a mailing address. It narrows the actual destination down to a specific region. The US post office groups the zip codes so that each number in the code narrows down the region a little further. The Internet works in a similar manner with network addresses. The router has a table of 'known' network id codes that identify where to forward each packet. The routers share these tables of information on how to reach different portions of the network among themselves. This information updates the routing tables of other routers and ultimately any host can reach any other host with the aid of the router network.

IP relies upon this routed addressing scheme where the actual address contains information on how to reach the distant host. An IP packet is a self-contained entity with both the sender's address and destination address encapsulated in the header. This equates to the connection-less model of networking where each packet contains both the data and delivery information. ATM does not require this additional per-packet overhead since the host places each packet into a preexisting virtual circuit. A cell only requires 5 bytes of overhead for every 48 bytes of data as compared to UDP/IP, which requires 40 bytes of overhead per packet [6].

2.2 DIS / HLA

The Distributed Interactive Simulation (DIS) system and its replacement the High Level Architecture/Run Time Infrastructure (HLA/RTI) are both wide area interactive simulation environments. This environment uses local and wide area networks to provide inter-simulation data transfer. DIS and the HLA/RTI require fast data transfer times, since the simulations are running in real-time. Inter-host communication delay must be less than 100 milliseconds for DIS [14]. Each simulation must be capable of communicating with any and all other hosts on the network. The simulations also require low computational overhead for network processing. Since the simulation is running in real-time, any excessive processor activity will cause noticeable skips and delays in the interactive simulation. These requirements dictate a very careful approach to the design and implementation of a network infrastructure.

2.2.1 DIS / HLA Using ATM

ATM provides for very low and fixed cell delays across both local and wide area networks. ATM also provides for low computational overhead since each cell is of fixed size and requires very little processing per cell. ATM provides for very precise data delivery so only the intended hosts receive the network traffic. This allows for very fast throughput at the host and can potentially reduce line traffic; allowing the use of inexpensive, lower rate leased lines.

2.3 Implementation of ATM

ATM is a collection of low level and intermediate level processes that allow for rapid configuration of local and remote switches. Much like a bank of phone exchanges, each switch can logically connect several physical ports together across a 'fabric mesh'. The fabric maps each incoming cell to an outgoing port. Since the ATM switch may map an incoming cell to several different physical outgoing ports, the switch must manage virtual connections within each physical port.

The overhead mentioned earlier is a combination of bits that identify which virtual connection a cell is moving along. As a cell arrives, the switch examines these bits of data and maps the cell to the corresponding outgoing physical port, and virtual channel. For instance, if a cell arrives from a host on physical port A of switch α with the number 15 as its virtual circuit, there will be a logical mapping of, say A:15 to D:102. The switch would then strip off the number 15 from the header of the cell, replace it with the number 102 and transmit the cell on physical port D. The switch that was at the other end of port D, switch β, will receive the cell on port C. As the cell comes in on port C, switch β will then map the cell to port B with a new header of 2. The mapping of (physical port): (logical circuit) to (physical port): (logical circuit) is the heart of ATM and requires something called signaling. Figure 2.1 shows a possible switch mapping.

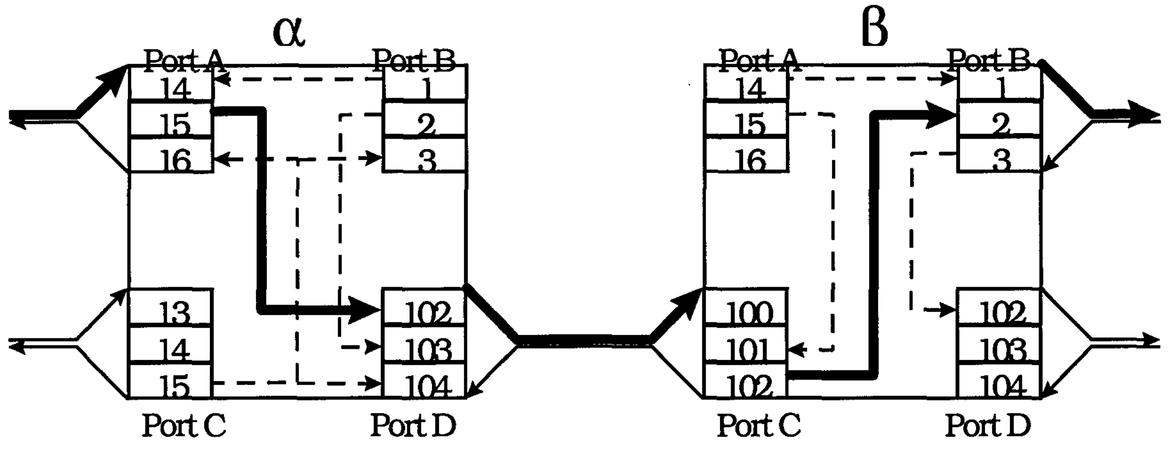

Figure 2.1: ATM switch interconnection

There are several things to note about logical connections within an ATM network. The connections are uni-directional (one way only). One port may have more than one destination (point-to-multi-point). The logical circuit of each port connects to either the transmitter or receiver of each physical port depending upon current call setup. An important consideration is that the cell address has significance only on the current physical port. Once the cell has been switched to the outgoing port, it will probably be assigned a new header address (virtual circuit number) based on the switch matrix.

2.3.1 Virtual Path Identifier / Virtual Channel Identifier

The numbers used as virtual circuit identifiers mentioned previously are actually a little more complicated within an actual ATM network, but they perform the same job. Each logical circuit is composed of a virtual path and a virtual channel within that path. Our virtual circuit number 15 for instance would actually be a number like 14:258 where 14 would be the Virtual Path Identifier (VPI) and 258 would be the Virtual Channel Identifier (VCI). There can be 256 VPIs per physical link and 65536 VCIs per VPI for a total of over 16 million potential circuits per port [6].

2.3.2 ATM Signaling

Signaling in an ATM network performs the same function as routing in a traditional network. It provides a road map from one host to another host, or one circuit to another using the VPI/VCI switch mapping. The 16 million connections per port may seem overwhelming but, in reality, the host can only use a fraction of these logical circuits. In most ATM interface ports, there is a limit of 1024 concurrent connections is due to memory constraints [5]. This limitation arises since each logical circuit must have a physical buffer in memory associated with it on the host. Another distinction of these logical circuits is that they are unidirectional. Since host to host data transfers are not usually equal, that is a host will not typically receive as much data as it sends, ATM specifies the link separately in each direction. This is important because each link may also include a data rate requirement. If a host expects to send (or receive) a significant amount of data, it can request a large data rate channel for the call. This allows the network to pick the most optimum path (for high data rates) available in the network. Signaling also allows the network to tell the host that the network can not handle that amount of data at the current time and negotiates a lower rate with the calling host.

2.3.3 Signaling Methods

There are currently several methods for signaling within ATM. These methods consist of processes that run either on the switch, the host, or both to help configure the switching fabric. This becomes the ultimate goal of signaling, to establish the inner VPI:VCI mapping of the switch. When ATM was first fielded, one goal was to replace the legacy Ethernet network currently in place. In order to do this, the LAN Emulation, or LANE signaling was developed. As ATM matured the ATM FORUM, a consortium of industry leaders provided another method of signaling known as User Network Interface (UNI). While LANE and UNI are not true equals, they both perform the same functionality and will be compared on a functional level.

2.3.3.1 LANE

LANE has several pieces; a client, a configuration server, an emulation server and a broadcast-unknown server (BUS) [5]. The client process resides on all interconnected hosts and performs all signaling communication for the host. The configuration server provides predefined mapping of which host belongs to which network. Since the switch can break down the physical network into several logical networks, this mapping creates a virtual LAN within a network. The emulation server provides an address mapping of all known hosts in the network. Each client has a predefined map of the configuration server and emulation server network address. Once a client has connected to the configuration server and the emulation server, the client can reach any other known host using the emulation server as a guide. The client uses the broadcast-unknown server when the emulation server does not know of a particular host, or when the client requests delivery to all connected hosts. The BUS basically redistributes the data to every connected host in the network.

2.3.3.2 UNI

UNI and the Network to Network Interface (NNI) work together to build up VPI / VCI connections in a wide area network. UNI is the local switch connection protocol and is most often referenced in standards publications [6], since it is the interface most applications deal with. UNI is a set of signaling commands that the host may give directly to the switch to establish connectivity. Within UNI there are subsets of servers such as the ATM Address Resolution Protocol Server (ATMARP) and the Multi-Cast ATM Server (MARS). ATMARP is basically like the name implies, it provides an address database that the client may query for addresses of local computer hosts. In Ethernet, a host simply requests another host's address using a broadcast. Since the broadcast does not exist in ATM, each host registers its address with an ATMARP server using a predefined connection. The client makes any future address requests directly to the ARP server. MARS provides a collection of addresses that have identified an interest with a specific Multi-Cast (mc) group. The mc group provides a method of mapping one transmitter to several receivers. This is like registering for a conference call. Anyone joining the call contacts the MARS to list their own address and then joins any current group call in progress. The ATMARP and MARS both function according to Request For Comment (RFC) standards publications established by the Internet Engineering Task Force (IETF).

2.4 Implementation of HLA / RTI

In order for a simulation to be realistic, it must have reasonably accurate knowledge of the simulated environment. This occurs when each simulated entity knows of all other pertinent entities. With current simulations having over five thousand entities, filtering must be performed or the simulations will grind to a halt just keeping track of each other [17]. The network filters data in HLA by an 'interest' expression. This expression reflects what the simulation can see, hear, touch, feel, etc. It can be a geographical region (a tank can see a two square mile area at position 'x'), a radio frequency, or any other logical division. The goal is to divide the simulation environment into manageable clusters so that each simulation does not need to know everything that happens within the entire simulation. An F-16 pilot, for instance, does not need to know about a grenade burst ten miles away. Thus each expression reflects a small piece of the overall simulation.

Simulations identify what parts of the battle space are important to them by subscribing to those expressions [12]. For example, if a user tunes a radio to frequency 102.4MHz, the simulation will request a copy of all radio traffic on that frequency. Similarly, if a tank moves over a hill, the simulation will join the group of simulations already in that area. Likewise, a simulation lets other simulations know of its own whereabouts by publishing to these identified groups. This mapping of what area, or expression, a simulation is interested in to the data available in that area is a crucial step in filtering out irrelevant traffic. Each simulation then knows exactly what data it needs and what data it is producing. The battle to reduce the amount of irrelevant traffic received by the host is almost won; the only job now is to get the network to cooperate with the hosts.

2.4.1 Current Strategies

The current networking strategy is to use an IP Multi-Cast (IPmc) group for each interest expression. IPmc relies upon the connection-less User Datagram Protocol (UDP) as the transport layer. UDP, unlike TCP, sends data when it is ready for transmission rather than bundling the data into efficient data packets. This reduces transmission time down to a minimum, but does mean more packets-per-second at the receiving host. The use of IPmc accomplishes several things. First, each host can subscribe and unsubscribe from these IP address groups at will. Next, the router network provides for a path to forward data to all users within an IPmc group. Finally, the network layer on each host will forward to the simulation host only the data that matches the host's current interest.

IPmc is a special case of the IP broadcast-addressing scheme. In an Ethernet environment, every host can physically 'hear' any data on the network, regardless of the data address. The host analyzes each packet's destination address to see if the address matches the host's address. Local Ethernet IPmc packets have a slightly different impact on the network from the wide area IPmc packets.

2.4.1.1 Ethernet IP Multi-Cast

A host can reconfigure its IPmc address very quickly, usually in a few hundredths of a second. The interface card must also update its Media Access Control (MAC) layer address to accept the incoming multicast addresses. The host can implement the MAC layer as either an explicit-accept (treat all multicast packets as broadcast) or as a programmable layer like the IP network layer. This allows the MAC layer to filter the data packets prior to delivery to the IP networking layer. Unfortunately the MAC layer can not filter large groups of addresses as efficiently as smaller groups. Each Ethernet interface card will have a table of current IPmc addresses at the network layer that the host belongs. The interface card will forward any packet with an IPmc destination address that matches one in the host's table up to the transport layer (UDP) for IPmc address confirmation. These IPmc address confirmations require host processing time, and can be CPU intensive since they require a CPU interrupt process.

2.4.1.2 Wide Area IP Multi-Cast

Processing IPmc packets in a LAN is simple, but routing IPmc traffic within a wide area network is rather complex. The first case, when the routers are unaware of IPmc traffic, occurs with older routers, which reject all IP broadcast traffic. To overcome this limitation, certain hosts ‘tunnel’ the information from one network to another. The tunneling process requires a host (D) on network 1 to forward all IPmc traffic to another host (D) on network 2. Once the host on network 2 receives the tunneled packet, it retransmits the packet as an IPmc packet on the local network. Figure 2.2 shows this process

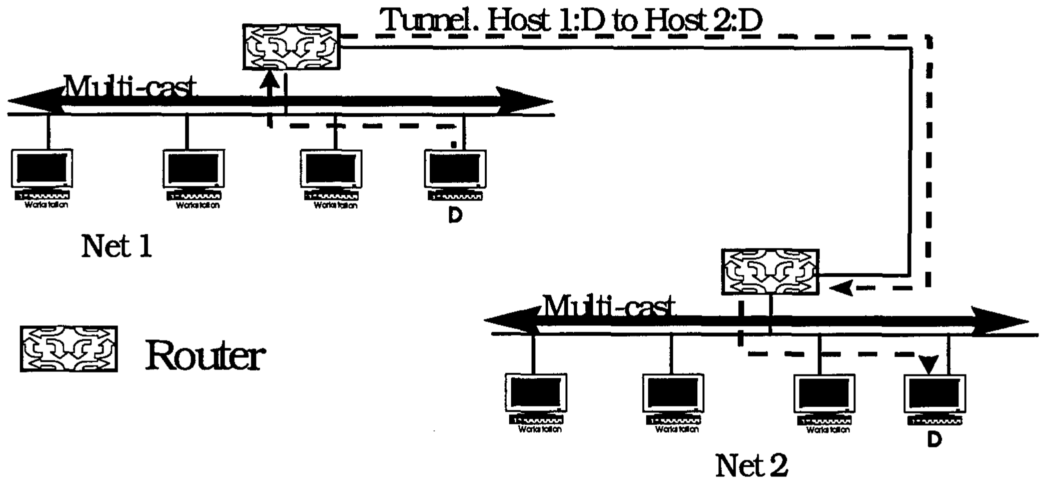

Figure 2.2: IP Multi-cast Tunnel

If the router is IPmc aware, each router will manage the IPmc groups via the Internet Group Management Protocol, IGMP [4]. The router periodically (every two minutes or so) queries all IPmc hosts on the local network for a listing of the groups to which the hosts belong. The router only requires one entry per IPmc address since all local network hosts receive IPmc traffic. If the router does not get a response for a certain address, it retires that group and any routes associated with it from the router's list. New groups become active by a host sending an unsolicited IGMP request to the router. As the router gets a request for a new group number, (depending upon routing implementation) it sends a request for the nearest upstream router that belongs to that group. Once the router finds the nearest neighbor belonging to the new group, it sets up a route for this new group. The router forwards any data generated locally while any remote traffic from that group to the local router is delivered to the local network.

2.5 Possible Areas of Improvement

The IPmc implementation has some network performance shortfalls. The first is if tunneling is used; a host has to become a communication server, which is a potential bottleneck to high-speed data. If an IPmc aware router is used, there is no way to immediately retire IPmc routes. The router must 'age' the route until it is removed. For frequent IPmc address hops this can create a large number of 'ghost' groups that generate useless network traffic. A third problem deals with the interface cards. Typically a host stores a multicast MAC address on the interface card as a condensed 6bit hash code [10]. This saves memory space and speeds up the address matching process, but if the host caches a large number of groups, the probability of an accidental match (not a true address) increases greatly. This inadvertent delivery wastes host-processing time. Also, as the number of cached addresses grow, the host must spend more time searching through the 'data base' of current IPmc addresses to weed out these false deliveries. One last problem is with the number of packets being delivered in a short period of time. As the number of Pk/s increases, the effect on CPU utilization could be dramatic.

2.5.1 ATM instead of IP

As described earlier, ATM is basically a point-to-point network. The UNI signaling process includes a point-to-multi-point service, but it is currently only for local network hosts [6]. There are several approaches to true wide area native ATM Multi-Cast (ATMmc), but each has significant drawbacks. Currently the ATM FORUM is investigating several options to standardize wide area Multi-Casting within ATM, but to date only an outline of problems that must be overcome has been drafted [1]. Current options for ATMmc (local and wide area) include using an N^2 mesh, using an ATMmc server, using an ATMmc tree-switch or waiting for future network development [6].

2.5.1.1 N-Square Mesh

The N^2 mesh approach requires each host to have a potential multi-point connection with every other host in the network, hence the name N by N connections. Each host can then distribute the data according to these interconnected mesh circuits. The drawback to this approach is that each address group will usually have a different make-up of hosts, so each group will require N^2 connections. Figure 2.3 illustrates the mesh approach.

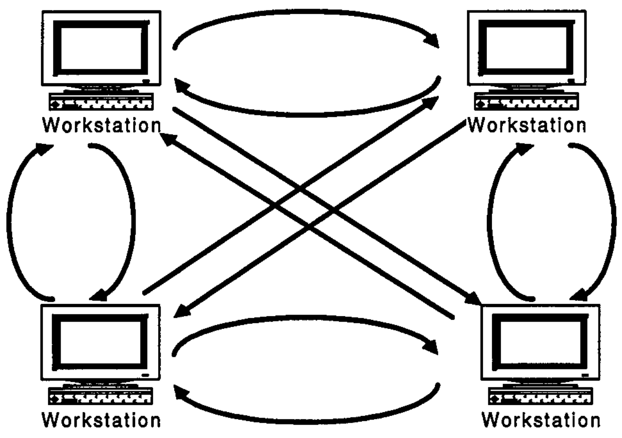

Figure 2.3: N^2 Mesh approach for 1 Group

The number of potential M-Cast connections for each host is on order of $G* N^2$ where G is the number of Multi-Cast (ATM or IP) groups. For the DIS / HLA network, this equates to almost 10 billion at 400 Groups with 5000 nodes [9]. Obviously a host will not use all connections, but the hard limit currently imposed by ATM interface cards is on the order of 1000. With the large number of virtual connections a large amount of duplicate data will be transferred over physical circuits. A pure N^2 mesh would be a poor choice for DIS / HLA for these reasons.

2.5.1.2 ATM Multi-Cast Server

The ATMmc server approach is similar to the tunneled IPmc approach. A designated host receives an incoming cell and retransmits the cell to a local group of hosts. Since only local hosts are involved, a limited number of locally managed ATM interconnections are required. A single host is typical, but a group of hosts may act as the server to distribute the data load and prevent bottlenecks. Figure 2.4 illustrates the ATMmc server approach.

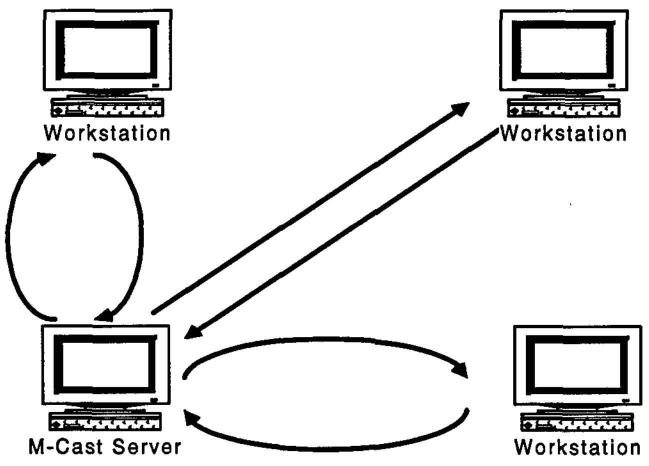

Figure 2.4: Multi-cast server approach

The drawback to the server approach is that a host could receive copies (reflections) of a message the host itself transmitted. This comes about when a host sends a packet to a group that it belongs. A filter process would be needed to discard duplicate messages. Another drawback comes from interfacing with the wide area network. Local hosts will know which servers belong to which groups, but getting data from a remote network host requires an inter-network database of servers and groups. This highlights the main disadvantage of ATM; that is it requires a rather sophisticated controlling mechanism for call management.

Another drawback is the sheer number of IPmc groups being used within DIS / HLA. Currently there are plans for around 10,000 IPmc groups. Even with a fraction of these being active at a host, the possibility of having an interest in more IPmc groups than the interface card can manage (about 1000) is a real risk. To help off set this, a strategy of overlaying more than one IPmc group to an ATMmc group could be developed.

2.5.1.3 Multi-Cast Tree

The ATMmc tree approach borrows the spanning tree functionality of a router. This process establishes a root node as a starting point in an actual wide area ATM network. As new hosts wish to connect; a tree is built out from the root to the new node. Figure 2.5 shows what an ATMmc tree looks like.

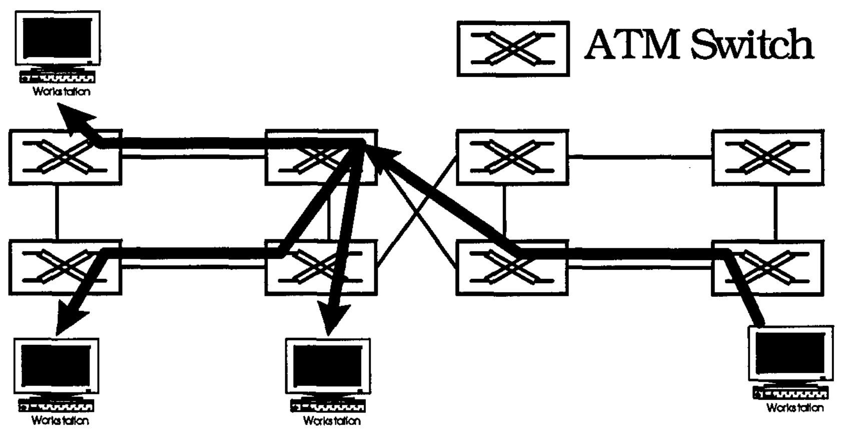

Figure 2.5:Multi-cast tree approach

The new node, called a leaf, is grafted into the tree at the nearest branch. This greatly reduces the amount of redundant data on the network, but it requires a host processor at each switching station to direct the growth and pruning of the tree. Again a complex management scheme must be used in order to harness the true power of the ATM network. Unfortunately, there are no current working implementations of the ATMmc tree process in a switch.

2.5.1.4 Current Research Efforts

There are current research activities trying to optimize which processes could be placed within an ATM switch to handle the ATMmc issues. For the tree approach, the grafting and pruning process could be handled automatically by a super set of current signaling. The switches could communicate route and path information on data flows to help optimize the overall virtual connections. The question is, how smart does the switch have to be? Routers started very humbly as bridging devices, but current routers require more computational power than many of the workstations on the Internet. Large supercomputers carry out current core routing processes just to handle the amount of address data and routing information [8]. ATM was developed to keep processing simple to avoid this pitfall.

2.5.2 Potential ATM Benefits over IP

Once the native ATM network has been built, several advantages over IP routed networks become apparent. The host interface card will only receive data that the host is interested in, this reduces the chance of inadvertent data. IP routing becomes more resource intensive as the number of IPmc groups increase. This is due to the increased amount of routing information that a router must process to keep track of the different groups. ATM does not have this routing information avalanche, but ATM does require processing time at startup to establish the set of interconnection links [6]. A connection oriented network layer will also provide for efficient ‘bundling’ of data to reduce the amount of CPU interruptions required for data delivery.

2.6 Realizable implementation

With all the identified tradeoffs and drawbacks to each implementation, can a 'best' approach really be reached? If so, is the cost required for that approach higher than the gain over the current scheme? To answer these questions, a study of DIS and HLA requirements needs to be performed to determine which implementation best matches the DIS/ HLA system's current and future needs. Once the requirements and implementation strategies are outlined, an analysis of how each strategy performs based on host processor usage, delivery time and resource utilization can be performed. With quantitative results, a cost versus performance study could be done to determine what resources would be needed to achieve a specific host performance increase.

2.6.1 Performance Metrics

The challenge with measuring a system's performance is coming up with meaningful statistics. Which metric has the greatest impact on the system? Will a cleaner data delivery pipe increase host performance, or will more data simply slow down the system? The first requirement is to establish a performance baseline of host processor usage based on IPmc data traffic. The host processor usage metric is being used to reflect how much processing time the host will use processing data as opposed to running the simulation. The main goal of the simulation is to provide a very high fidelity, real time interactive environment to the user. Once this baseline is established, the native ATM host processor usage can be measured to determine what, if any, performance gains there are. These measures will need to be conducted in an isolated local network setting to ensure a fair comparison.

2.7 Summary

The main objective of this thesis is to evaluate the practicality and performance of introducing native ATM in the DIS/HLA network for local data distribution. There are no current practical methods for using native ATM for wide area distribution, as outlined in section

2.5, so a local network application is the most obvious place for evaluating this approach. The most significant challenge is to synthesize a method for mapping IPmc groups to ATMmc groups. This method must reduce the IPmc groups (order of 10,000) down to a small number (order of 500) of ATMmc groups without losing any data and must maintain integrity of the wide area network.

3. Approach

Current developmental efforts to optimize simulation performance rely heavily upon IP multi-casting to provide packet filtering. The ability to use the network to 'filter' unwanted data and deliver only what is required is a definite plus, but current multi-casting techniques are limited in the number of IP multicast (IPmc) addresses that can be implemented. This limitation is a result of the memory limits of the routers and LAN switches as well as the limited number of IPmc addresses that the host interface card can effectively filter. A router's limit is around 1,500 IPmc addresses and switches' limit is about 3,000 IPmc addresses [3]. Another limitation imposed by IPmc addressing is that each message is a distinct unit, since each message probably falls into different addressing groups. With each message being sent as a distinct packet, the messages may hit the Pk/s limitation of the simulation host, effectively blocking any additional data communication with the host. This limitation is currently around 5,000 packets per second [12].

3.1 Areas of Research

This thesis has three main areas of research. The first studies the effects that larger IPmc groups have on host CPU efficiency. As the number of IPmc groups increases, more CPU time will probably be required to manage and filter these groups. The second area of research examines the Pk/s issue. The focus here is on three key components that make up this limitation, namely the hardware platform, the LAN interface card and the operating system. The various impact of each component allows for a maximizing approach to increase the actual Pk/s that a host can receive. The third area of research is the study of a proposed method for overcoming the IPmc address and packet per second limitations. This method is to use a series of hosts to first filter the IPmc data traffic and then deliver the data to the simulation hosts in efficient message bundles.

3.1.1 Definitions

Several definitions are presented here to establish a sense of order. These definitions keep a sense of perspective when comparing current concepts with the studies and approaches developed later in this document.

3.1.1.1 Packet

A packet is the set of data presented to the networking media for transmission. A packet may contain one or more distinct data sub elements.

3.1.1.2 Packet Address

The packet address is the address placed on the packet for physical network delivery. A packet address and IP network address are the same. Typical packet addresses follow the "234.2.5.1" dotted decimal notation and may be unicast, multicast, or broadcast. The message address generates the packet address by dividing the message address by the number of messages per IPmc address. Typical simulation packet address ranges are from 300 to 2000 distinct entries [16].

3.1.1.3 Message

A message is a unique simulation data entity. The message is variable in length, but always contains a message address field that is currently two bytes in length. The typical message is 150 Bytes in length (commensurate with current DIS standards).

3.1.1.4 Message Address

The message address is from the 'subscribe / publish' address of a message and is well defined across all simulation hosts. This address space is established for all simulation exercises. With a 2byte address, the message addresses range from 0 to 65535 with no priority flag or may range from 0 to 32767 when using a priority flag. Increasing the number of bytes allocated to the message address will increase the address space. Obviously the message address comes from a much larger range than the packet address, so we must convert between the addresses. Currently, the packet address is the message address divided by 32. This provides for a packet address range from 0 to 2048.

3.1.1.5 Bundle

A bundle consists of a two byte bundle mask, from 1 to 255 messages, an indexed field identifying the ending position of each message and the last byte containing the message count of the bundle. Figure 3-1 shows the structure for the bundle.

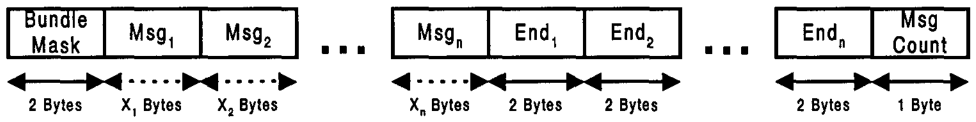

Figure 3-1: Bundle structure

3.1.1.6 Bundle Mask

The bundle mask is a 2byte source-identifier. The bundle mask has the two most significant bits set on with the least most bits set to indicate the converter number. Each local host has a unique mask where only one bit is on. Host one for example, has the mask (hex) 00 01. The third host has a mask (hex) of 00 04 up to the sixteenth host with a mask (hex) of 80 00. The host mask prevents messages from being accidentally transmitted back or 'reflected' to the sending host.

3.1.1.7 Host Group / Group Address

A group is the logical association of hosts expressing an interest in, or 'subscribing' to a particular message address. Each message address therefore has a corresponding host group associated with it. These host groups are dynamic, since a simulation host may subscribe and un-subscribe to any group at any time during the simulation. The group address is simply the

2byte representation of the sixteen hosts within the group. Table 3-1 illustrates the converter database.

Table 3-1: Converter Database

Message address	Group address
21	205 (11001101... in binary)
22	45 (00101101... in binary)
23	223 (11011111... in binary)

The binary representation is the real key to the database. Each bit represents an interest flag for a particular host within that group. For message address 21, the first two hosts (from left to right) subscribed and have a code of 1. The next two hosts have no interest and have a code of 0. As each host joins a group by subscribing to that message address, the field bit position for that host becomes to a one. As a host leaves a group, the bit position changes to a zero. This process accommodates up to 16 hosts using a two-byte address field. A larger field will accommodate more hosts. To facilitate message filtering at the simulation host, the converter process removes the actual message's address and replaces it with the group address during bundling. As the messages are received, the hosts simply look for its 'bit' in the group address of a message to determine if the message is meant for that host.

3.1.2 Local Area Network versus Wide Area Network

The reason the focus of this study is on the local network instead of the wide area network is the issue of scale. While Wide Area Networks use ATM it requires a VC for every data path. If host-to-host or LAN-to-LAN multicast interconnections are attempted using ATM, the physical number of VC's required is more than either the host interfaces or the ATM switches can handle. Even if ATM VC's are established and managed on the WAN, the signaling mechanisms have to approximate what is being done with Protocol Independent Multicast (PIM) which can handle IP traffic as well as ATM. VC call setup and tear down also introduces 'per call' delay and traffic. As each host changed their multicast group associations, multiple VCs have to be established or released, creating exponential (N^2 Mesh design) signaling traffic and delays.

3.1.3 Required Changes within DIS / HLA

The Message-to-Bundle converter is a set of software processes running on either one or multiple hosts. These hosts act as gateways between WAN message traffic and local bundled traffic. This allows the WAN to stay 'one message per packet' based IPmc, since this appears to be the most optimal method for wide area traffic. With the small change of adding the message address to HLA systems, full data interchange is possible, since all LAN traffic is converted to IPmc messages for transmission, and from IPmc messages when received. The change to legacy systems involves placing the 'subscribe to' message address as a two byte header within the message itself. This may seem redundant, since the message address generates the actual IPmc address of the packet. Unfortunately, there is no way to recover this address from the IPmc packet upon receipt. The only way for the converter host to know the message address is if the address is part of the message itself.

3.2 Study 1, IP Multicast Address Effects

The first study, the IPmc test, is for one type of host and one network media. This limitation focuses on the impact of the networking card alone. The only computers available in this study to test IPmc efficiency were the Ethernet SGI hosts. This limits the variety of testing, but the results are indicative of standard Ethernet interface cards. The test runs a series of network performance evaluations on the SGI Indigo2 (see Section 3.3.5 for exact host descriptions) that is receiving a variety of IPmc network traffic. The host subscribes to a percentage of the available IPmc addresses. A sending host transmits IPmc traffic uniformly distributed across the entire test range of available IPmc groups. The uniform distribution allows for the tests to be reported as time average CPU utilization rates. If we used another distribution, the utilization rates would have to be converted to averages to provide for a meaningful comparison. The test keeps the size of the packets identical, as well as the actual number of packets per second being delivered to the host. This ensures that the only deviation in CPU utilization is the actual process of filtering IPmc addresses from among the arriving data. The subject host runs a timed loop of floating operations while receiving the IPmc traffic. We measured a floating operation per second or FLOP performance level from this loop. Each test had an additional run using the SGI's native 'osview' performance indicator. While the 'osview' does not provide as precise a measure as the FLOP test, it is still useful to corroborate the FLOP performance test.

3.3 Study 2, Packet-Per-Second and Byte-Per-Second Rates

The second study delves into the area of network performance known as 'packet-limited'. This area comes about when the media could provide additional data to the host, but the host is incapable of processing additional packets from the network. In order to fully understand the elements that influence packet-limited performance, several areas of the network architecture are evaluated.

3.3.1 Network Interface Card / Media

The first area to be evaluated is the interface card / media component. We expect this component to have some impact on performance, but because NIC cards must be interchangeable among PC platforms, it is doubtful if one vendor could significantly optimize an interface card beyond what the standard architecture allows. Two SGI hosts, an Onyx, and an Indigo2, are configured with both a 10Base Ethernet and a 155Mbps ATM interface card to evaluate the media effects. Identical tests will determine if the ATM card with its superior data rate can provide more packets per second than the Ethernet card. This test uses 1-byte UDP packets to ensure that the media bandwidth does not influence the outcome.

3.3.2 Hardware Platform

The second evaluated component is the host hardware. It is expected that this component has a significant impact on the actual number of Pk/s that can be processed since the hardware dictates such metrics as bus and CPU speed, interface architecture and other physical optimizations. To evaluate the hardware platform's influence, three host's running the same version of the Linux operating system with identical interface cards are tested. The first host is an Intel 486 33Mhz system, the second is an Intel 100Mhz Pentium system and the third is a 200Mhz Intel Pentium Pro system. The researcher understands that the Linux operating system is compiled for each host individually, and therefore does not provide identical operating systems but does provide a common reference. The native Linux performance meter validated the netperf results. A test to evaluate the effect of multiple CPUs on network performance is run on the single CPU Indigo2 and a quad CPU Onyx, both using the 250Mhz R4000 processors.

3.3.3 Operating System

The final component is the actual host operating system. While the operating system contains the code that handles interface card interrupts, we doubt it can provide a substantial increase in the number of packets the host can process. To evaluate the effects of the operating system we analyzed the results from the hardware test to determine if there were any differences between a host running Linux and the same host running Windows 95.

3.3.4 Study 2, Test Environment

The Pk/s test used 'netperf', a relatively standard Internet network performance tool. The Pk/s test used the UDP protocol with packets of one byte to ensure that the media bandwidth did not create an artificial limit. In order to measure the theoretical bytes-per-second that a host can process, the netperf test uses the internal loop-back port for testing. The internal loop back test isolates the host from the interface card and from the media. This loop back test allows testing of the actual CPU and internal bus and is the only way to get fair hardware and operating system results. As with the FLOP test, the native performance meter software confirmed the netperf results.

3.3.5 Network Lab Setup

There are actually three labs used to conduct the tests. The graphics lab has three 250Mhz R4000 Indigo2's, two quad 250Mhz R4000 Onyx's, a 200Mhz R4000 Indigo Challenge and a quad 195Mhz R10000 Onyx2 connected via a 10BaseAUI Ethernet segment. Two of the Indigo2s, the two Onyx hosts and the Indigo Challenge are also interconnected by a FORE ATM switch. The SGI's are running Irix version 6.2, use the NFS process and have a standard complement of background processes running. Since other researchers use the labs at AFIT it is impossible to strip a network down to a 'pristine' condition. The results are more indicative of real world situations and therefore the background processes are left running. This introduces variability to the tests, so each test is performed a minimum of five times with at least three of the results in a 3% margin of error for validity.

The second lab is the PC lab, composed of the following computers: an Intel 486 CPU at 33Mhz, an Intel Pentium CPU at 100Mhz, and an Intel Pentium Pro CPU at 200Mhz. A 10Base2 LAN interconnects these computers which run both Linux and Windows 95 operating systems. The third lab, the Zoo, is an FDDI-based ring of 25 Sparc20 hosts running the SunOS 4.2 operating system. The Zoo also uses NFS with a standard set of background processes left running.

3.4 Study 3, Suggested Implementation - The 'Bundle Converter'

The goal of this implementation is to reduce the number of Pk/s presented to a simulation host during processing while maintaining, or increasing the actual number of messages delivered to the host. We expect that reducing the number of Pk/s that a host must process will free up a significant amount of CPU processing capacity.

3.4.1 Implementation Requirements

As a minimum, the implementation must provide for the same level of performance and communications capabilities currently in place within the HLA simulation environment. This dictates that the simulation software and networking hardware does not require any modifications to be inter-operable. To this end, the bundle converter must:

- Receive 'native' IPmc message packets from the WAN router
- Timely deliver messages to the simulation host (less than 100ms delay)
- Accept bundled message traffic from the simulation host and deliver to the WAN and other simulation hosts as required
- Prevent packets from being 'reflected' back to the originating host
- Update IPmc tables at the WAN router via IGMP control packets

3.4.2 Additional Capabilities

By implementing a layer three programmable device into the LAN environment, several additional capabilities become realizable. Below are just a few of the additional capabilities provided by this implementation. Further study and optimization efforts may show even more capabilities, but for this effort the following additional capabilities will suffice:

- Reduce CPU utilization by bundling the message packets
- Extend message address space from 2,000 to over 65,000
- Establish flow control and quality of service

- Provide for reliable delivery
- Exceed the 5,000 message per second limitation of the simulation host

The first benefit reduces CPU utilization. While there is a certain amount of processor time wasted as it sifts through unnecessary message traffic within a bundle, we expect that the CPU processor savings provided by reduced network interrupt handling will compensate for this wasted time. Even so, this bundling method has a 'break-even' point in terms of when the bundling process pays for itself. The break-even point will be measured in Pk/s. If a host routinely exceeds this point in normal HLA traffic, it is a good candidate for the bundling process.

The second benefit is the extended message address space. Conventional IPmc subscription regions have a limit of one to two thousand addresses, depending on the networking components. The next generation of IPmc routers and switches boast on the order of three thousand IPmc addresses. Much like sub-netting breaks one IP network into several smaller networks, the bundling process subdivides each IPmc address into zones. An analogy is the cache memory of a computer system. The larger resolution of the IPmc network will contain several related subscription regions. As a host moves among these related regions, the converter simply adds and drops the host from the local IPmc group address. There is no need to subscribe and wait for the wide area network to 'catch-up' since the converter already receives the necessary data.

A third benefit is the element of flow control and quality of service. This occurs since the converter knows exactly how much data is being transmitted to the wide area and local area networks. The converter passes congestion data back to the hosts, which in turn limit the amount of non-critical data they are transmitting. If congestion is only at the wide area interface, the converter employs 'source quench' and reduces the amount of wide area non-priority traffic.

Another benefit is the element of reliability on the local network. Since a message can have a priority tag, the converter and hosts can track it and send acknowledgments even though the User Datagram Protocol (UDP) is an unreliable protocol. Unfortunately we cannot implement wide-area reliability with the converter process, but with flow control the priority messages have a better chance of being delivered.

One final benefit is that the simulation hosts are capable of sending and receiving more than 5,000 messages per second. Under the current methodology, each packet contains one message. The physical Pk/s limitation forces the message per second limit. Overcoming this physical limitation is possible with an arbitrary number of messages per packet.

3.4.3 Converter Implementation Design, Software Process

The design is broken into three main components: a set of level 1 converters, a central converter, and a set of host interface processes. The architecture allows for up to six first level converters labeled C_1x, where x is replaced by a, b, c, up to f. There is only one central converter, labeled C_2. The C_2 processor can support up to 16 hosts. The host interface processes are labeled H_1, H_2, through H_16. Figure 3-2 shows the overall design with packet interchange format.

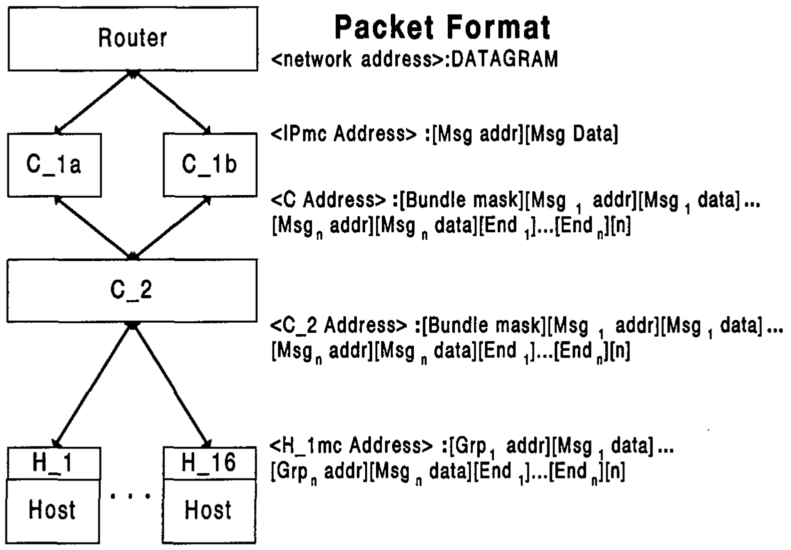
Router
Packet Format
<network address>:DATAGRAM
C_1a
C_1b
<IPmc Address> :[Msg addr][Msg Data]
<C Address> :[Bundle mask][Msg 1 addr][Msg 1 data]... [Msg n addr][Msg n data][End 1]...[End n][n]
C_2
<C_2 Address> :[Bundle mask][Msg 1 addr][Msg 1 data]... [Msg n addr][Msg n data][End 1]...[End n][n]
H_1
Host
H_16
Host
<H_1mc Address> :[Grp 1 addr][Msg 1 data]... [Grp n addr][Msg n data][End 1]...[End n][n]

Figure 3-2: Overall converter process

3.4.3.1 Router interface, C_1

The router interface accomplishes several functions. Its primary job is to receive and transmit message-packets to the router. The C_1 converters must also advertise IGMP message traffic to the WAN, but the C_1 converter must be careful so that the C_1 converters do not overlap addresses. If the addresses were to overlap, another C_1 converter would inappropriately receive one C_1 transmission. The three processes identified in the flow chart run in parallel on the C_1 host. Figure 3-3 shows the software flowchart for the C_1 process, which listens for messages from the router. A temporary buffer stores the message and the message end point. The C_1 host checks for the congestion level specified by the C_2 host. As the congestion approaches 80 % of the media's capacity, the converter may drop some low priority messages. The converter sends high priority traffic regardless of congestion.

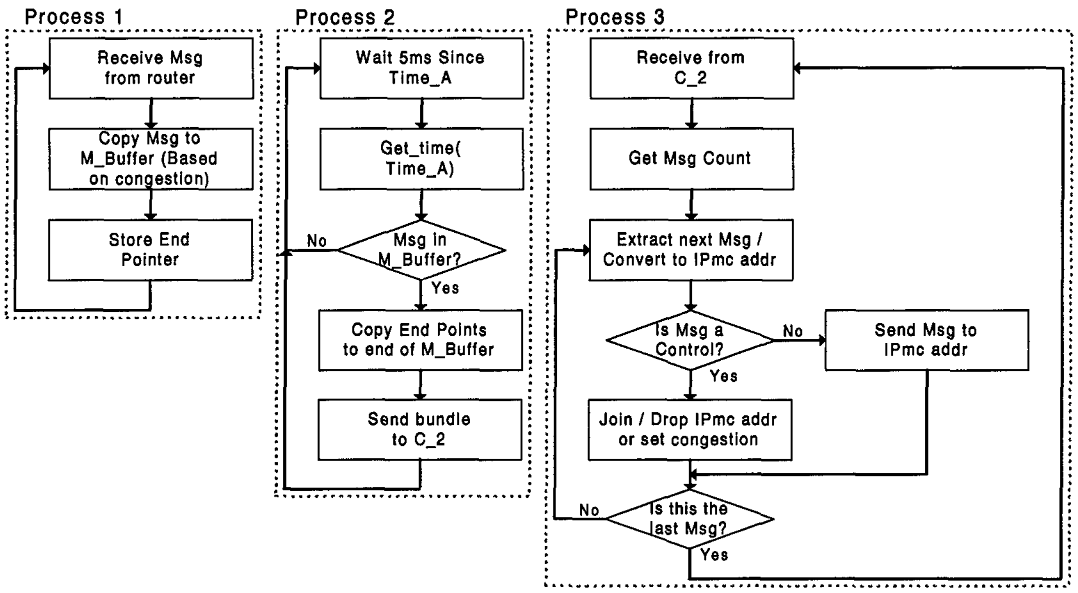

Figure 3-3: C_1 flow chart

The second C_1 process waits for a 5millisecond timer to expire and then checks to see if the converter bundled any messages. If so, then it appends the bundle buffer with the end point identifiers and number of messages in the bundle, then sends the bundle to the C_2 host. The third process listens for traffic from the C_2 host. When a bundle arrives, the C_1 host unpacks the bundle and processes the messages. The C_1 converter changes each message address to an IPmc address for transmission. If the message is data, it is simply transmitted to the router. If the message is a control message, the C_1 host must either send out an IGMP message or update its congestion level.

3.4.3.2 Host interface, H_1

The host interface has the primary job of direct simulation connectivity. Since the simulation provides the host interface with bundled message traffic, the host interface must be the bundle to message interpreter. Figure 3-4 shows the software flowchart for the H_1 process.

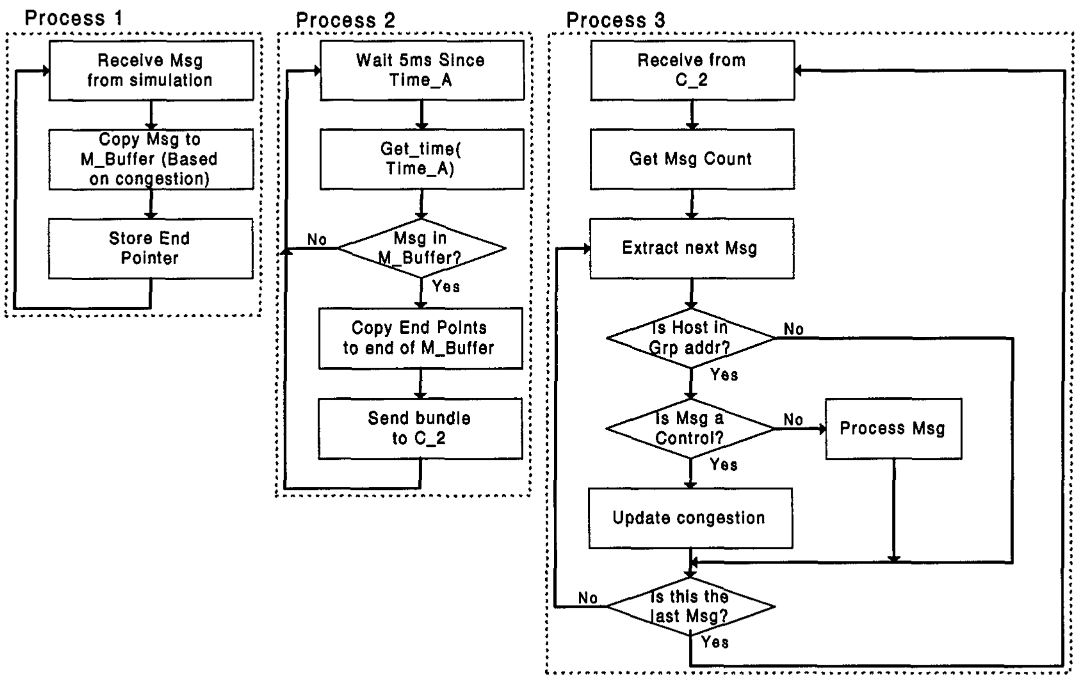

Figure 3-4: H_1 flow chart

The host process, H_1, is very similar to the C_1 process, since it has much the same functional requirements. The main difference is that the simulation is the message source instead of the router. A control message from the C_2 process indicates that a congestion level has changed. The H_1 simulation also initiates all message address joins and drops via control messages. These control messages drive the C_2 and ultimately the C_1 hosts to respond by updating the local router or switch to start sending the IPmc traffic.

3.4.3.3 C_2 process

The C_2 processor is the central communications and control point within the converter system. Since all data passes through the C_2 converter, it keeps track of how much data is being moved and calculates potential congestion problems based on bandwidth usage. The C_2 process is the most complex and requires a more careful description. There are multiple C_1x send processes, where the x would be replaced by an a, b, etc., corresponding to the number of C_1 converter hosts being used. The C_2 host can segregate the message traffic bound for the C_1 hosts since each message will be destined for only one C_1 converter for transmission.

There is only one H send process, since the message transmissions to the simulation hosts occur as a broadcast, and all simulation hosts receive the broadcast. These C_1 send and H send processes are very similar to those described earlier.

The receive-bundle process is the controlling function within the C_2 converter. It also controls the other C_1 converters. The process receives a bundle, extracts the message count / bundle mask, and determines where the bundle came from. If the bundle is from the C_1 converter, it must contain message data for the simulation hosts. Therefore the C_2 process extracts the messages, looks up the group address, and sends the message to the host-send buffer, based upon the message priority and congestion. If the bundle is from a host, then the messages can be either control or data. If the message is a control message, the host will either join or drop a message address. When a host wants to join a group, the C_2 converter adds that host into the C_2's group address database. There can be more than one message address associated with an IPmc address. Therefore, if no other host is receiving message traffic from the corresponding IPmc address, the C_2 host must tell the least used C_1 host to request the IPmc address. If there is a C_1 host already getting IPmc traffic for that address, the C_2 host does nothing. Figure 3-5 shows the software flowchart for the C_2 process.

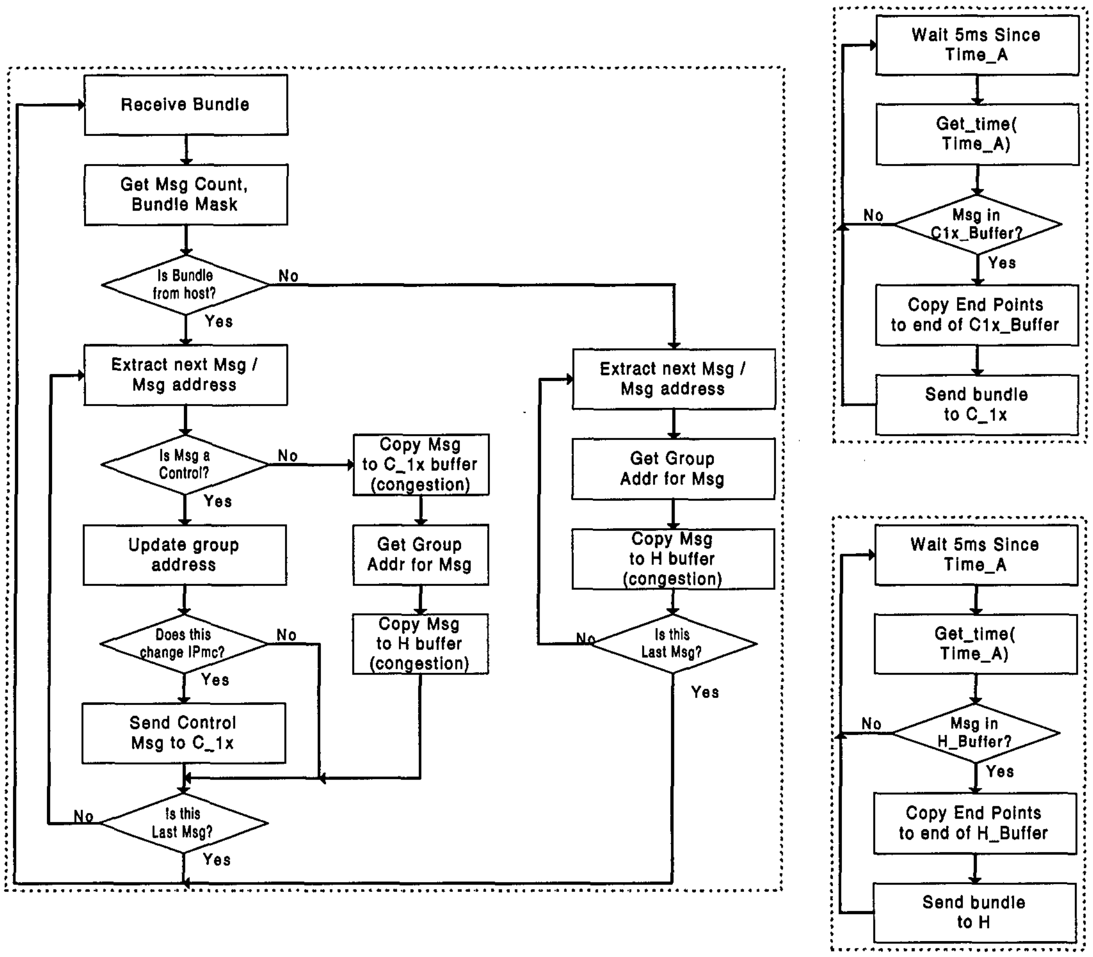

Figure 3-5: C_2 flow chart

If the simulation host wants to drop a message address, the C_2 host then removes it from the message group address and checks the other groups belonging to the IPmc address for that message address. If no other simulation hosts want traffic corresponding to that IPmc address, then the C_2 host tells the appropriate C_1 host to drop that IPmc address.

If the message contains data, the C_2 process copies the message to the appropriate C_1 send buffer and looks up the group address to see if a local simulation host wants that message. If so, the C_2 host looks up the group address that determines the local simulation hosts that will receive the message. The C_2 process then removes the sending host's bit from the group address to prevent reflection. The process then replaces the message address with the modified group address and copies the message and address to the host send buffer for local delivery.

3.4.4 Topology, Data Flow for Converter Process

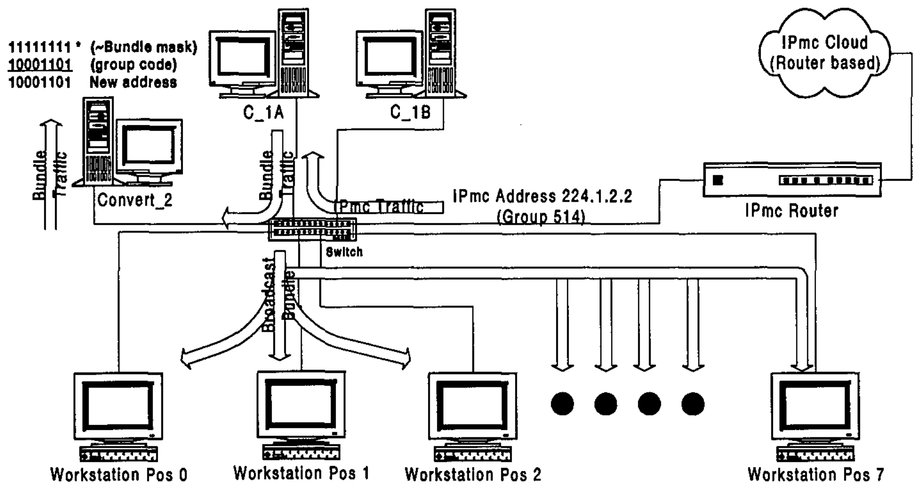

Figure 3-6: IPmc message traffic

Figure 3-6 shows the stages of data flow for the converter. The converters labeled C_1A and C_1B are the 'front line' converters. Their job is to receive and transmit message traffic to the wide area network. One benefit of using multiple C_1 hosts is that they can aggregate the traffic throughput across their interface points. Two C_1 hosts can handle 10,000 Pk/s; three hosts can handle 15,000 Pk/s and so forth. Much like combining four 64k circuits into one 256k circuit to reduce queuing delays, this aggregate Pk/s performance reduces the amount of time a message spends in the system queue. There are two drawbacks to using a bundle converter system. The first drawback is the obvious need for more hardware. Fortunately, a Pentium class machine running at 100Mhz or better should satisfy the Pk/s and data rate requirements. The other drawback is that each additional level-one converter imposes another 400 Pk/s of data (200 Pk/s from C_1 to C_2 plus 200 Pk/s from C_2 to C_1) on the C_2 converter. Therefore, the number of C_1 converters used is limited to a maximum of about six to eight, depending upon the number of simulation hosts serviced.

The level-one converter receives all IPmc message traffic associated with it and performs a preliminary bundle on the data. It is important to spread the data load equally across the C_1 converters to maximize their Pk/s rate. The C_1 converter receives bundles from the C_2 converter and transmits them as individual messages. Since each level one converter is responsible for a distinct set of IPmc addresses, no cross talk from one converter back to another occurs.

3.4.5 Topology, Configuration Cost

Figure 3-6 shows a general topology for the converter process. There are three specific topologies evaluated in this study. The common elements in the topology are the converter hosts. Since each converter host is a Pentium-class machine with very little graphics or disk storage requirements, we assign the nominal cost per converter at $1,000. This is a very reasonable estimate for common Intel Pentium machines today. The variable elements within the topology are the interface cards and the LAN hub. The converters will have 10/100Mbps interface cards installed.

3.4.5.1 Baseline topology

The baseline topology is the standard configuration of six simulation hosts interconnected by a standard 10Base Ethernet LAN with a router as the WAN interface. The cost of the baseline topology is no cost, or zero dollars for comparison. Figure 3-7 shows this topology.

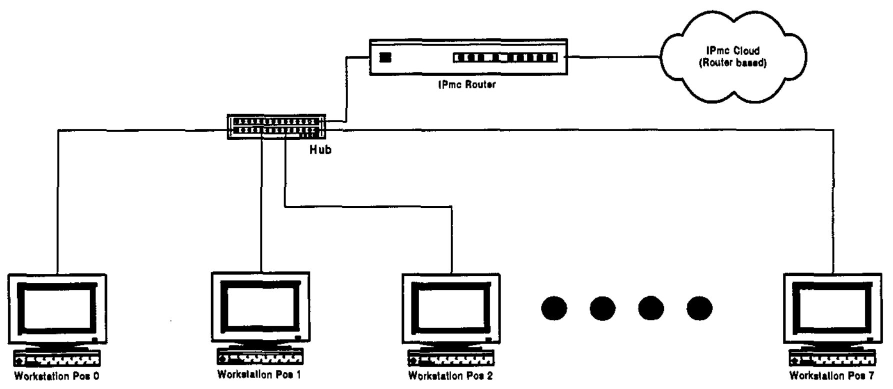

Figure 3-7: Baseline topology

3.4.5.2 Switched Baseline

The change for the second topology is to simply replace the 10Mbps shared media with a 100Mbps switched media. This demonstrates one of the possible future upgrades to improve network performance. The cost for this topology is $2,000 for a 100Mbps switch and $300 per host interface card. Total cost is $5,000 for a 10-host system. This has the same physical and simulation topology as the baseline topology with the hub replaced by a 100Mbps switch.

3.4.5.3 Baseline With Converter

This topology introduces the converter hosts to the baseline network. The baseline media is the 10Mbps shared Ethernet. All other components remain the same so the cost for this topology is $3,000, which is simply the cost of the three converters. Figure 3-8 illustrates this topology.

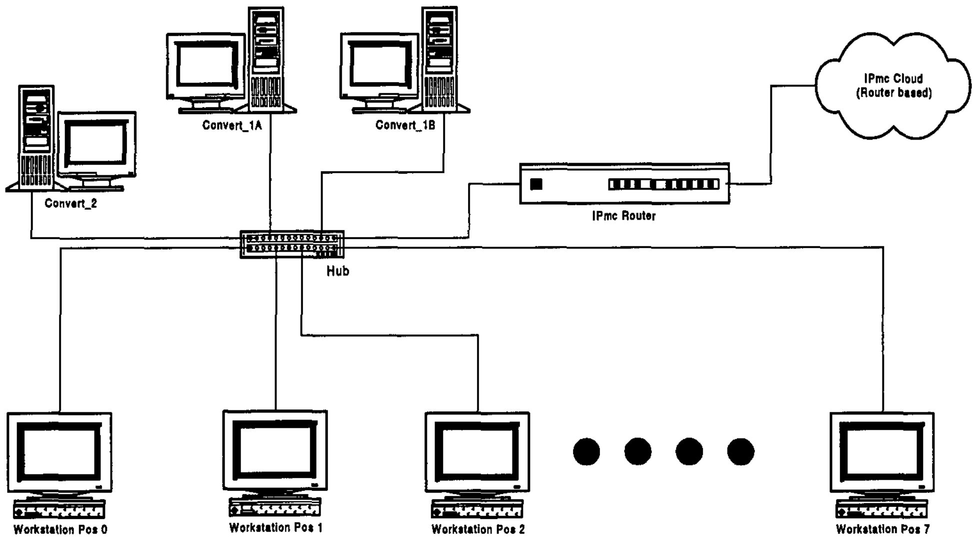

Figure 3-8: Base converter topology

3.4.5.4 100Mbps/10Mbps Switched converter

The next topology consists of a 100Mbps Switch for the converters, 100Mbps interface cards on the converters and 10Mbps cards on the simulation hosts. The additional cost here is for the four port 100Mbps switch, which has a nominal price of $1,500 bringing the total cost to $4,500. Figure 3-9 illustrates this topology.

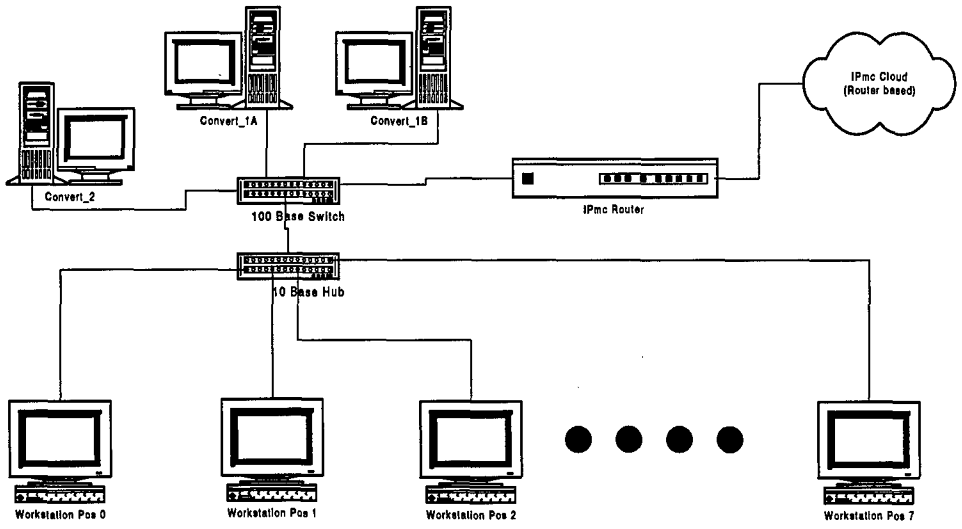

Figure 3-9: 100Mbps/10Mbps switched converter topology

The third topology changes the 10Mbps hub to a 16 port 100Mbps hub interconnecting the simulation hosts which are retrofitted with 100Mbps interface cards if necessary. The switch and hub both have an expected price of $1,500 while the interface cards are about $300 each. This brings the total cost to $9,000 for a 10-simulation host network. This has the same physical topology as the 100/10 switched topology with the 10Mbps hub changed to a 100Mbps hub.

3.4.6 Test loading

The actual tests require a variety of data loading to evaluate the expected and actual performance gains. There are seven test cases. The first three tests have each host originating 200 messages per second and receiving 1000, 2000 and 3000 messages per second, respectively from the WAN. In all tests, each simulation host sends its data to two other random hosts in the network. Host originated data is 'inside' traffic while WAN originated data is 'outside' traffic. The last four tests have the simulation hosts originating 450 messages per second and receiving 4000, 5000, 6000 and 7000 messages per second, respectively from the WAN. These last four tests measure extreme data delivery cases.

3.5 Network analysis, modeling and simulation

Since the makeup of the networking lab here at the Air Force Institute of Technology is relatively fixed and under constant use, simulation software will expand the converter process beyond what is physically realizable. These simulations utilize the data gathered from the physical tests to include processing usage per packet and per byte, processing requirements for the actual converter software and network characteristics of several media / component combinations. These simulations explore the impact of new technology such as fast Ethernet, multicast aware switches and mixed media networking topologies on both the conventional and bundled networking approaches.

3.5.1 Simulation software and design

COMNET III, Version 1.3 is the network simulation software used to evaluate configurations and data loading that are physically unrealizable here at AFIT. The test cases that can be physically duplicated are analyzed to make sure that the simulations accurately depict real world results. The simulation design closely follows the actual operation of the converter process. Standard COMNET models simulate the media, switch, hub and router components. The hosts are customized for packet and data CPU usage to measure the data loading impact on CPU performance as well as data delay. The baseline topologies mirror the physical topologies as closely as possible.

3.5.1.1 Simulation Design, Baseline and Switched Baseline

The baseline simulation design is a router, six simulation hosts and a 10BaseT Ethernet hub (see figure 3-7). The router has a message source that generates 150 Byte all-host multicast packets with an exponential inter-arrival time. While we expect that DIS and HLA WAN traffic is not for all hosts, this approximation provides for even data loading and reduces the simulation run times. A random distribution can be used, but the test results have to be reported in average Pk/s loading to provide for a meaningful comparison. This testing focuses on average delays and CPU usage, not specifically the variations so the test is performed on this average basis. The hosts generate 150 byte multicast packets destined for two random hosts with an exponential inter-arrival time. The switched baseline uses the same model as the baseline with the hub replaced by a switching component. Figure 3-10 shows the COMNET representation of this topology.

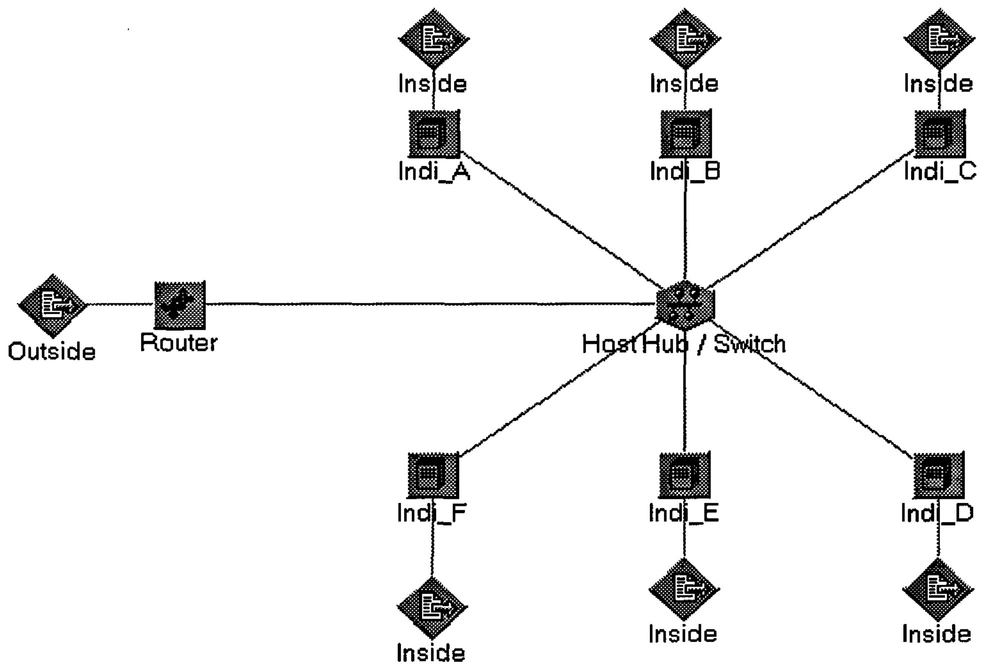

Figure 3-10: COMNET topology for baseline

3.5.1.2 Simulation design, Converted baseline

The converter augmented baseline simulation has several processes that reflect actual design functionality. The outside message source is identical to the baseline, as are the media and router modules. The C1x_Bundler process listens for outside messages and then writes them to a zero delay disk file to emulate a memory buffer. Every 5ms the C1x_wl process reads this disk file, sends that data on to host C_2 and then erases the file. The C1x_lw process listens for bundles from the C_2 host and then transmits them to the router using a modified UDP protocol. In order to get the model to accurately send out 150 byte messages, the maximum transmission unit, or MTU for the modified UDP protocol was set to 150 bytes. This forces the C_1x processor to break down the large bundle traffic into 150 byte packets, just as the real converter will do. This is important, since CPU interrupts occur when the host sends packets as well as when the host receives packets. At host C_2, the C_2wl process listens for bundles from the C_1 hosts and writes them to a file called 'host'. The C_2lw process listens for bundles from the host and copies them into the host file as well as a file called 'wan' since the process must send these messages to the router as well. Every 5ms the rebundle-in process reads the host file, broadcasts this data as bundles to the simulation hosts and then erases the file. Every 5ms the rebundle-out process reads the wan file, sends the data to a uniformly distributed random C_1 host and then erases the file. The actual C_2 host will attempt to keep the data load as evenly distributed among the C_1 hosts as possible so the uniform distribution is a good approximation. Every 5ms the inside message source creates a random number (zero to four) of 150 byte messages, bundles them up and sends them to the C_2 host. Figure 3-11 shows the COMNET model.

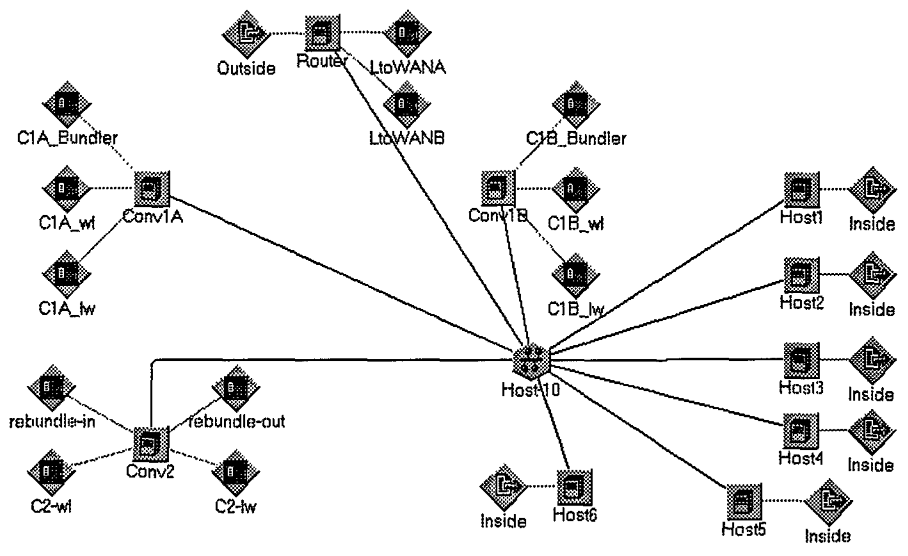

Figure 3-11: COMNET topology for converted baseline

3.5.1.3 Simulation design, Switched converter

The switched converter has the same modeling components as the switched baseline, only the media has been changed to reflect the addition of the 100Mbps switches and hubs. Figure 3-12 shows the COMNET topology for the switched converter.

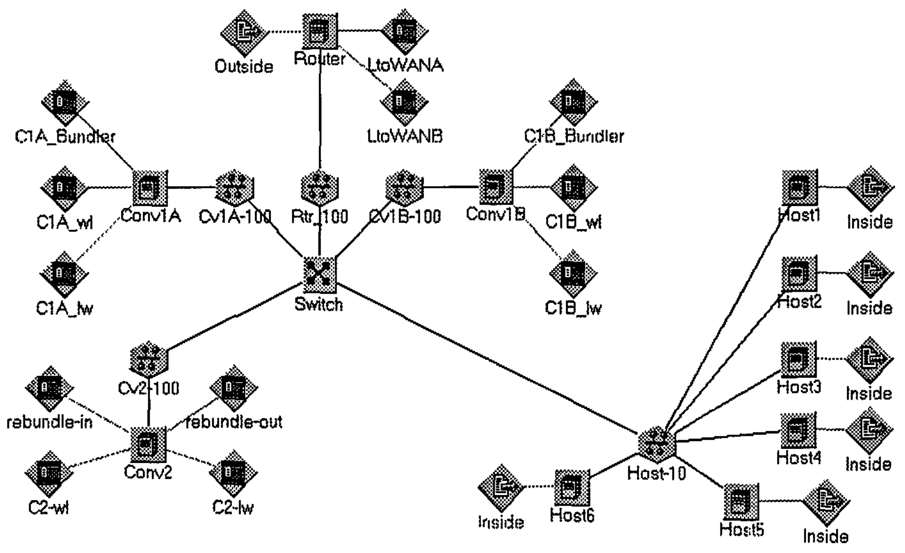

Figure 3-12: COMNET topology for switched converter

3.6 Conclusion

The converter process basically uses a broadcast for data delivery to the hosts, forcing each host to receive and filter all message traffic by group address. However, it does allow for the bundling of messages and transmission to the hosts at 200 bundles per second. We expect that the filtering process will be very cheap, in terms of CPU usage when compared to the usage of network interrupts [13]. Since current HLA traffic is approaching 2,000 Pk/s, we expect that almost half of the CPU's capacity is spent processing packet interrupts. As the number of Pk/s rise due to higher fidelity models and increased number of hosts, we expect that simulation hosts will be unable to handle the volume of message traffic.

As indicated earlier, the interrupts created by the network interface card creates a heavy load on the CPU. The message to bundle converter provides a method to reduce the network interrupts at the host, help reduce congestion and increase the available subscription space for the simulations to use.

4. Results

4.1 Overview

The results obtained from this research show that the distinction between packet limited (unable to receive additional data due to the Packet-per-second (Pk/s) limitation of the host) and byte limited (unable to receive additional data due to media bandwidth) can be exploited to yield higher data throughput and lower CPU utilization without excessive delays. The use of UDP as a data protocol, combined with a programmable network layer device provides for a reliable, low latency, efficient, priority driven data delivery process that is a significant improvement over current methods. The following individual results are from three specific tests; IP multicast efficiency, packet size efficiency and the converter process.

4.2 IP Multicast efficiency

The IP multicast test revealed that the host operating system does use the CPU to filter IP multicast packet addresses from the pool of subscribed addresses. As the interface card accepts a packet from the network, the card performs an initial hardware hash check to determine if the packet 'could be' for this host. If the packet clears the hash check, the O/S kernel receives the packet for processing at the IP layer. The first action the IP layer process performs is another address check to ensure that the multicast packet is for this host. If the packet IPmc address matches the host database, the IP process unwraps the IP layer and submits it to the UDP layer for further processing. Our tests confirm that as the numbers of address groups increase beyond 50 to 100, the O/S receives a larger number of packets where the interface card inadvertently accepted the packet. As noted earlier, this is due to the limited hash space the interface card has.

Figure 4-1 illustrates the percentage of CPU time taken to process the packets. Each category has a constant data reception rate from 200 to 2,000 Pk/s as shown in figure 4-1. At 2,000 groups and 100% reception rate, the CPU utilization is higher than if data used a broadcast address, since the CPU checks each multicast packet for address relevance. At 5,000 groups and 80% request rate (1,600 Pk/s host load), the CPU usage is higher than the 100 group, 100% request rate (2,000 Pk/s host load) even though the load is lower. This indicates that there is a diminishing return when using IPmc as a filtering mechanism.

The CPU performance loss is a function of both the total groups and the number of requested groups from that pool. For instance, if the number of requested groups remains constant, say 40 or fewer, then the interface card can still efficiently filter the traffic even though the total group population may be very large. Typically though, the number of requested addresses increase as the available address space increases, and this is where CPU performance degrades quickly. For this reason, a limit of between 200 and 1,000 total group addresses appears to be a good choice. Another factor is how much irrelevant IPmc data is present on the LAN. Figure 4-1 demonstrates a sharp increase between the 200 and 500 group case at the 20% request rate indicating an increase in irrelevant IPmc traffic. This degradation stems directly from the interface card's inability to filter out this unwanted data without relying on the O/S IP stack. A curve traced along the path of about 20 requested addresses shows where the degradation starts. The 10% interest line increases sharply at 200 group addresses. The 40% interest line also sharply rises at the 50-group address point. This indicates that the interface card's optimal filtering range is at or below 20 requested addresses.

The degradation in CPU performance is due to the IP protocol stack in the host but the network media may affect the IPmc degradation. When using IP over ATM, the CPU probably would not experience this degradation since the interface card would only receive traffic over the ATM virtual circuits corresponding to the requested IPmc addresses. Unfortunately there is a limited number of ATM virtual circuits available for multicast mapping. The ATM interface will run out of virtual circuits at about 500 IPmc addresses. Another option is to use an Ethernet switch that is capable of understanding the Internet Group Management Protocol (IGMP) to add and remove IPmc addresses from the Ethernet port. Current IGMP switches can handle about 3,000 multicast addresses per switch. This will significantly reduce the amount of unwanted packets being processed by the interface card, but the host must still filter the packet to ensure an IP address match. This IP layer filtering could waste up to 5% of the CPU's processing time, as in the case of the 10,000 IPmc groups at 100% interest level. Even though the host subscribed to all traffic (100% case) and the data load is constant, the CPU is at 35% usage for 10,000 groups but only about 30% for 10 group. Table 4-1 and Figure 4-1 shows the actual performance measurements taken during the test.

Table 4-1: CPU utilization, IPmc test

	Total number of groups transmitted at 2000 Packets / Sec									
Rcvd packet / sec (% of groups)	10	20	50	100	200	500	1000	2000	5000	10000
2000 p/s (100%)	30	30	30	30	30	31	30	31	33	35
1600 p/s (80%)	26	27	27	28	28	29	28	29	30	33
1200 p/s (60%)	23	23	23	26	26	26	26	27	27	29
800 p/s (40%)	19	20	19	22	24	24	24	24	25	26
400 p/s (20%)	16	16	16	17	19	22	22	22	22	23
200 p/s (10%)	14	14	14	15	16	19	21	21	22	22

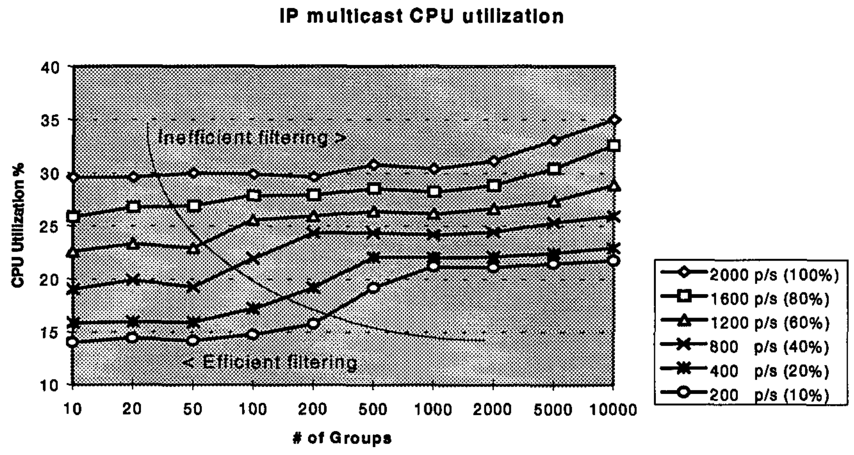

Figure 4-1: CPU utilization, IPmc test

Another point to consider is the dynamics of the network. One area noted during the testing was the time required to process an IGMP message. A simple loop test showed that the host could only send out about 1,500 IGMP messages per second. The reason for this limitation is not clear, but it most likely comes from the table updates on both the interface card and the host IPmc address tables. Although it is not expected, if a large enough group existed in an extremely dynamic environment, the host's IGMP messages may have trouble keeping up with the application, or significant IGMP delays could occur.

4.3 Packet size and packet-per-second rate efficiency

The packet size and packet-per-second (Pk/s) rate studies showed that the media, host platform and even host operating system play a large role in determining capability and efficiency. One interesting thing about UDP is the collapse of data throughput. This happens because a sender can typically generate data faster than the receiver can physically accept it. In TCP, the software flow-control mechanism simply adjusts the window size, or number of outstanding packets, to throttle back the source. In UDP there is no flow control so the sender can virtually drive the receiver into overload, corrupting data currently in the receiver's buffer. In the Ethernet environment, the throughput collapse tends to be gradual, since packet collisions occur more often at the higher data rates where throughput collapse occurs, creating an artificial flow control system. In the ATM network, the collapse is very pronounced and usually occurred between 7,500 and 8,000 Pk/s.

4.3.1 Packet size results presentation

4.3.1.1 Media impact on CPU utilization

The first analysis focused on the impact of the actual network media on the CPU utilization. The study compared identical hosts using Ethernet and ATM as the network media. The test host was the Indigo2. The results where there are no data points occur when data throughput collapse occurred. For the Ethernet case this usually occurs at about 9.0 (± 0.5)

Megabits per second as expected since the media bandwidth is 10 Megabits per second. Collapse begins to occur in the ATM network as the host approaches 90% CPU utilization. Table 4-2 and 4-3 show the CPU utilization for both the ATM and Ethernet network media under various packet sizes and rates. By comparing these two tables we see the ATM media is between 1 and 25% more efficient, in terms of CPU utilization, than the Ethernet, depending upon the packet rate and size. While the maximum improvement is impressive, it occurs when the CPU has already reached over 60% utilization. When the data rates are below 1,000 Pk/s, about 10% is the maximum improvement with around 4% being the average. These tests show that the actual CPU utilization improvement realized by converting to an ATM will be very dependent upon the underlying network traffic.

Table 4-2: CPU utilization, Indigo2 Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkt/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	3	4	3	4	5	6	6
500 p/s	12	11	12	13	12	13	15
1000 p/s	17	19	24	24	26	28	
2000 p/s	30	33	44	45	47		
3000 p/s	41	51	53	54			
4000 p/s	54	63	76				
5000 p/s	67	80	90				
	Max Packet / Sec		7000		**Max bits / Sec***		116Mbps

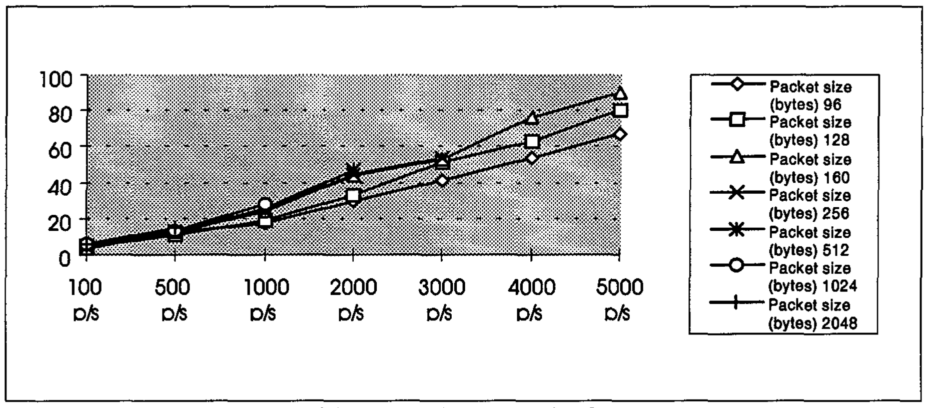

Figure 4-2: CPU utilization, Indigo2 Ethernet

Table 4-3 reveals that throughput collapse for the ATM network only occurs when the CPU approaches 100% utilization with a maximum data rate of 142 Megabits per second. Since the underlying media is an OC-3 (155 Megabit per second) data channel, this shows that the CPU is actually the bottleneck in this case. While data throughput is higher than the Ethernet case, the maximum Pk/s rate is actually lower. The cause for this is unknown, but the test showed that when the ATM interface received packet rates higher than 5,000 Pk/s, the interface would reject about 10% of the first 1,000 packets. The interface became stable after this initial loss, but only as long as the Pk/s rate was constant. Any change in the Pk/s rate produced another 10% packet loss. For this reason, the ATM network is stable only to 5,200 Pk/s.

Table 4-3: CPU utilization, Indigo2 ATM

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	2	2	2	2	2	2	2
500 p/s	14	14	15	16	14	15	14
1000 p/s	11	13	15	13	18	30	34
2000 p/s	23	24	25	28	34	39	55
3000 p/s	35	40	44	45	55	60	85
4000 p/s	52	53	55	66	77	82	
5000 p/s	58	62	64	68	90		
	Max Packet / Sec		5200		**Max bits / Sec**		142Mbps

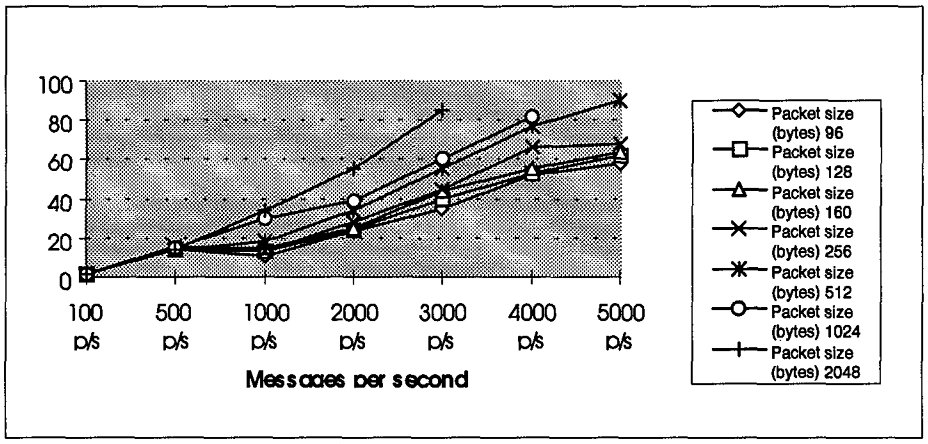

Figure 4-3: CPU utilization, Indigo2 ATM

These two cases show the assumption that packet processing would be comparatively more CPU intensive than byte processing. Figures 4-4 and 4-5 show CPU utilization rate to packet size instead of CPU utilization rate to packet rate. Since both size and rate affect the overall data throughput, this comparison shows how each parameter impacts CPU utilization. In the previous charts, a steep increase is evident as the Pk/s increase from 100 to 5,000 Pk/s. A plot of CPU utilization versus packet size shows only a moderate increase is evident. The most critical observation is a comparison of specific data throughputs on both charts. In Figures 4-2 and 4-3, we see that by increasing our throughput by an order of magnitude using a higher packet rate we experience a five to ten fold increase in CPU utilization. The same order of magnitude increase in data throughput by changing the packet size (Figures 4-4 and 4-5) has virtually no impact at data rates less than 500 Pk/s, and at most a three fold increase at 1,000 Pk/s. The most dramatic example of this is at 100 Pk/s and 2048 Bytes per packet where the CPU is being used at two and six percent for the ATM and Ethernet, respectively. The same data rate (200 Kilobytes per second) at 96 Bytes per packet requires 2,200 Pk/s; at this same throughput rate the CPU is being used 23% and 30% for ATM and Ethernet, respectively.

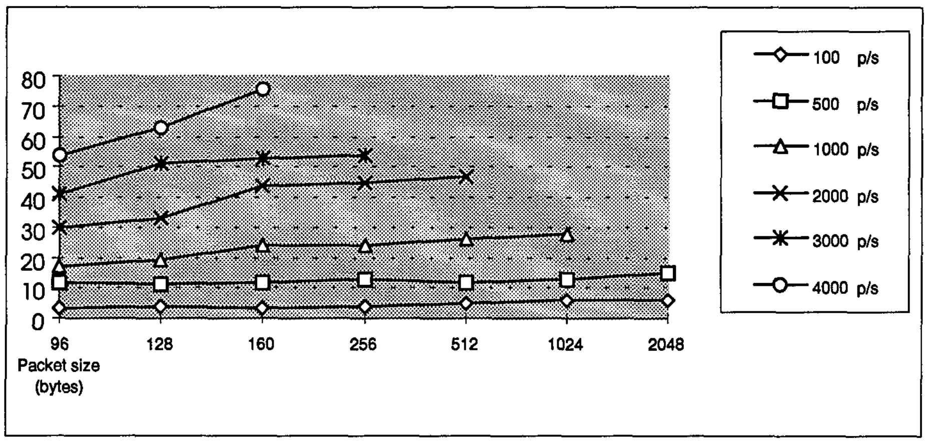

Figure 4-4: CPU utilization vs. Packet size, Indigo2 Ethernet

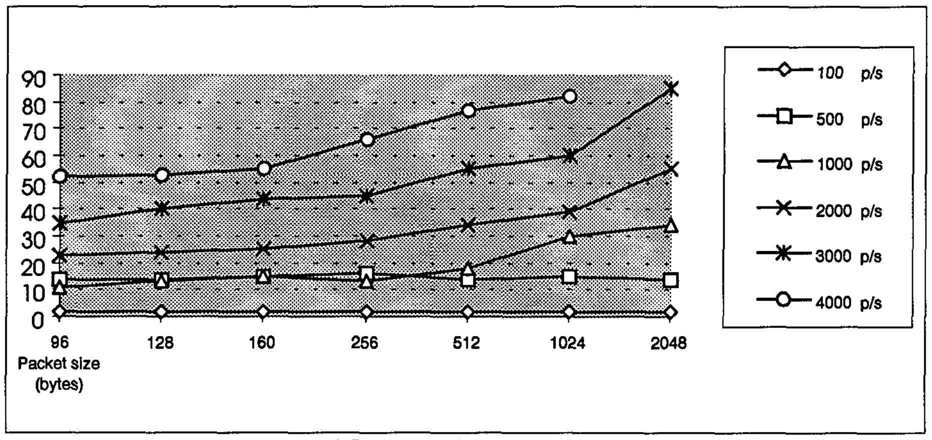

Figure 4-5: CPU utilization vs. Packet size, Indigo2 ATM

By increasing packet size and reducing packet rate, the data throughput remains constant, but the CPU utilization can be reduced by up to an order of magnitude. This was common among all platforms, media and operating systems tested.

4.3.1.2 Host impact on CPU utilization

A second series of tests analyzed the impact different host platforms could have on network performance. The two test platforms were a 60Mhz 586-based system and a 200Mhz 686-based system. Both computers used the Red Hat Linux version 4.2 operating system. The tables show the difference the host platform can make on the CPU efficiency, especially at higher packet rates. The 686 platform had the highest packet per second processing of all hosts at over 15,000 Pk/s. In fact, even an Onyx2 could not generate enough packets to overload the interface card. While the 686 platform had a much higher Pk/s rate, it still could not handle the higher data rates that the SGI computers could (22Mbps for the 686 as compared to 140+Mbps for the SGI). Tables 4-4, 4-5 and Figures 4-6 and 4-7 show the 586 and 686 system results, with the absence of data again indicating a collapse in data throughput.

Table 4-4: CPU utilization, 586 Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	3	3	4	4	6	9	18
500 p/s	9	11	12	16	26	58	91
1000 p/s	18	20	23	31	51	89	
2000 p/s	35	39	41	59	99		
3000 p/s	51	59	65	87			
4000 p/s	67	77	86				
5000 p/s	83	95					
	Max Packet / Sec		6600		**Max bits / Sec***		12.6Mbps

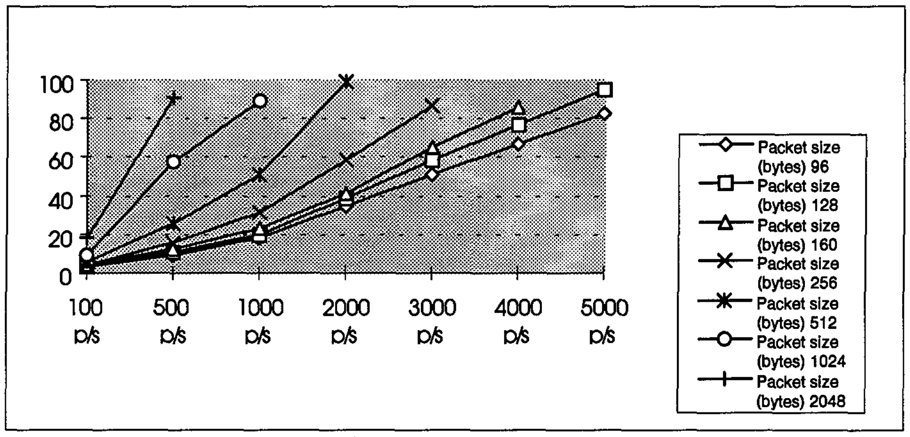

Figure 4-6: CPU utilization, 586 Ethernet

Table 4-5: CPU utilization, 686 Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	96	128	160	256	512	1024	2048
100 p/s	3	3	4	6	12	16	21
500 p/s	8	14	18	22	32	38	45
1000 p/s	15	18	21	26	46	54	
2000 p/s	21	23	26	45	58		
3000 p/s	41	42	45	54	75		
4000 p/s	44	48	58	66			
5000 p/s	54	64	66				
	Max Packet / Sec	15000+		Max bits / Sec*	22.5Mbps		

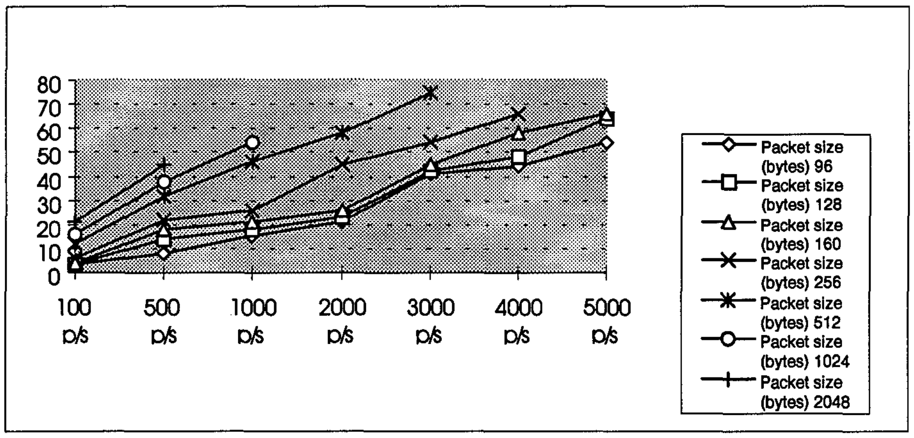

Figure 4-7: CPU utilization, 686 Ethernet

4.3.1.3 Multiple processor impact on CPU utilization

The multiple processor tests utilized the ATM networking media. The results from the multiple CPU tests indicate that the quad R4000 Onyx system could provide CPU utilization performance gains depending upon the packet rates. At moderate to lower data rates (less than 2,000 Pk/s), the quad system proved to be only 5% to 10% more efficient. The osview performance meter indicated that only one of the four CPU's handled network interrupts. The largest gain in efficiency over single CPU systems was around 40% at 3,000 Pk/s and 2048 bytes per packet. This shows that while multiple CPU systems have an advantage, they can not provide a true 1 to 1 processor to performance increase. The multiple CPU systems cannot exceed the Pk/s or bit per second performance of the single CPU system. The quad processor system was unstable in the ATM case at 100 Pk/s rate, therefore rates are not reported. In the 100 Pk/s case the CPU utilization rates were uniformly spread from two to five percent across all packet sizes. The quad system could sustain 5,400 Pk/s with the ATM media, but as explained earlier, only when the packet rates were constant. Tables 4-6 and 4-7 show the comparative performance levels of the single and multi CPU systems.

Table 4-6: CPU utilization, single processor Indigo2 ATM

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	2	2	2	2	2	2	2
500 p/s	14	14	15	16	14	15	14
1000 p/s	11	13	15	13	18	30	34
2000 p/s	23	24	25	28	34	39	55
3000 p/s	35	40	44	45	55	60	85
4000 p/s	52	53	55	66	77	82	
5000 p/s	58	62	64	68	90		
	Max Packet / Sec		5200		**Max bits / Sec**		142Mbps

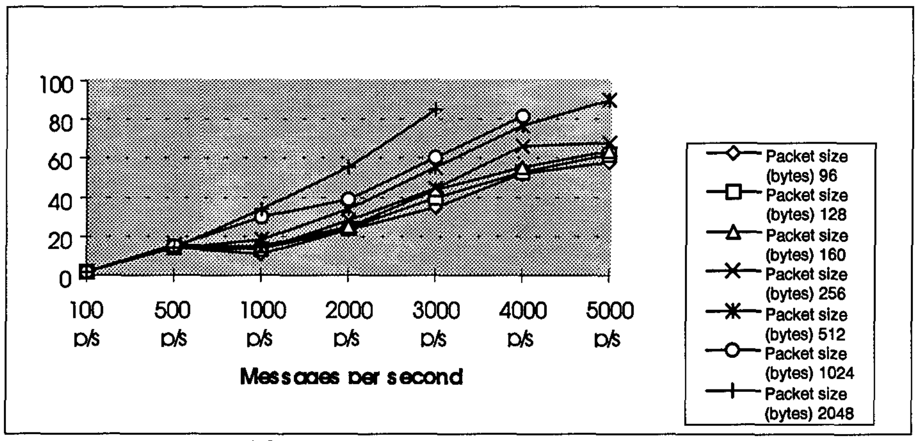

Figure 4-8: CPU utilization, single processor Indigo2 ATM

Table 4-7:CPU utilization, multi processor Onyx ATM

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	96	128	160	256	512	1024	2048
100 p/s							
500 p/s	15	15	16	18	16	17	18
1000 p/s	21	21	22	24	23	22	27
2000 p/s	35	35	35	36	37	36	36
3000 p/s	45	44	45	45	47	44	44
4000 p/s							
5000 p/s							
	Max Packet / Sec	5400		Max bits / Sec	141Mbps		

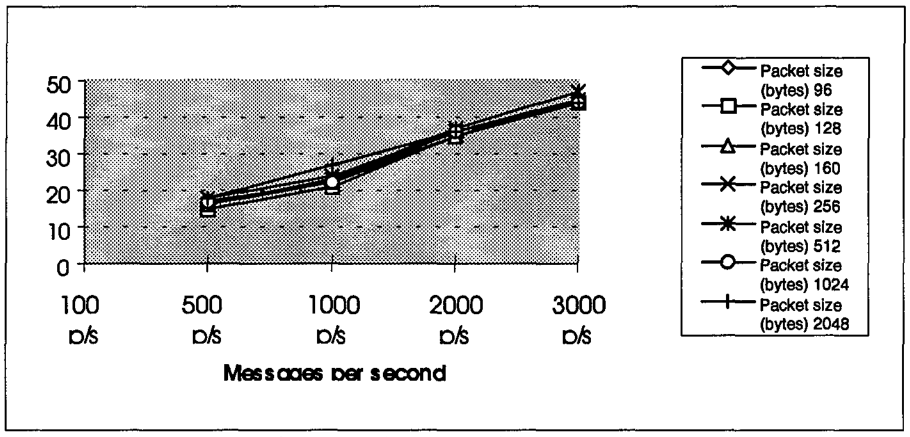

Figure 4-9: CPU utilization, multi processor Onyx ATM

4.3.1.4 Host operating system impact on CPU utilization

The assumption that the operating system would have little impact on CPU utilization was proven wrong. The test proved that the exact same computer could have significantly greater performance by simply changing to a more efficient operating system. The reason is that the operating system must handle each network interrupt. A specific set of instructions must be executed for each received packet. Depending upon the way the vendors construct the interface drivers, this set of instructions may become rather extensive. The first operating system, Windows 95, is built to be highly inter-operable with different hardware. This inter-operability requires extensive layering to allow each third-party vendor to supply the required interface drivers. The Linux operating system however, is compiled into an executable kernel for each specific machine and for each specific interface card. The host operating system also impacted the maximum Pk/s rate. The operating system did not have a significant impact on the maximum data rate, since the actual hardware bus speed determines this. Tables 4-8 and 4-9 illustrate the performance ratings for the Windows 95 and Linux operating system study. The absence of data indicates where the CPU's reached the 100% threshold.

Table 4-8: CPU utilization, 486 Win 95 Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	24	26	26	29	32	38	50
500 p/s	53	54	56	66	75	89	
1000 p/s	84	86	88	100			
2000 p/s	100						
3000 p/s							
4000 p/s							
5000 p/s							
	Max Packet / Sec	2500			**Max bits / Sec***	12.6Mbps	

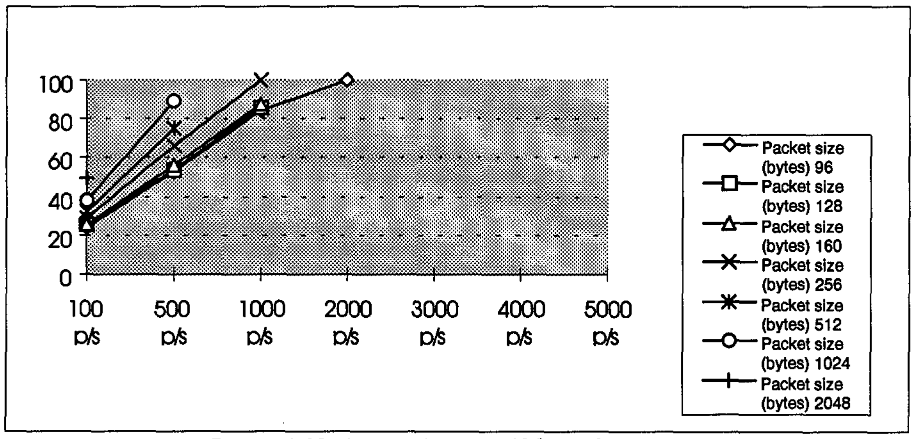

Figure 4-10: CPU utilization, 486 Win 95 Ethernet

Table 4-9: CPU utilization, 486 Linux Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	8	8	8	9	9	11	21
500 p/s	19	21	22	25	35	48	98
1000 p/s	32	34	38	45	63	93	
2000 p/s	57	63	68	85	100		
3000 p/s	80	89	94				
4000 p/s							
5000 p/s							
	Max Packet / Sec	3500		**Max bits / Sec***	13.5Mbps		

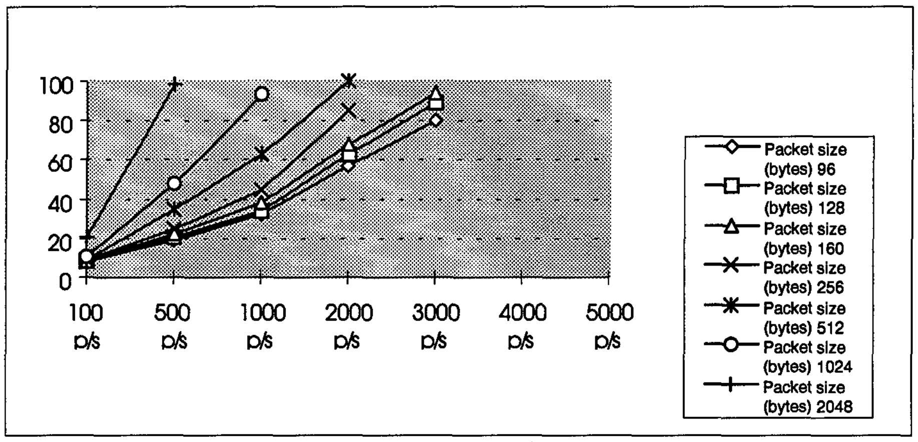

Figure 4-11: CPU utilization, 486 Linux Ethernet

4.3.1.5 IP versus AAL5 impact on CPU utilization

The data protocol, either IP or ATM Adaptation Layer 5 (AAL5), had a rather unexpected impact. The networking industry has often criticized ATM for being less efficient than IP in its data handling. While this is typically true for bandwidth utilization, it is not always true of CPU utilization, as this test shows. The AAL5 protocol is much less efficient at lower data delivery thresholds. It appears this protocol has a handicap of 10% CPU utilization with no data throughput. As the data rates increase beyond 4,000 Pk/s, the CPU utilization rates turn back to AAL5's favor. Unfortunately data loss starts to occur in the ATM network at 5,000 Pk/s, so it is unknown if this trend is useful at extremely high packet rates. ATM has a larger maximum transmission unit (MTU) of 9188 bytes / packet versus Ethernet's 1500, allowing for more data in each packet. The larger MTU allows for the very high data rates (150Mbps) that the ATM network can provide. Ethernet would theoretically reach a maximum of 5,000 packets at 1,500 bytes each or 60Mbps if the host bus interface and media provided the necessary bandwidth. Tables 4-10 and 4-11 compare the results of the two protocols using the ATM network.

Table 4-10:CPU Utilization, Indigo2 ATM-IP

% CPU utilization rate as a function of Packet size (bytes)

Pkt/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	3	4	3	4	5	6	6
500 p/s	12	11	12	13	12	13	15
1000 p/s	17	19	24	24	26	28	
2000 p/s	30	33	44	45	47		
3000 p/s	41	51	53	54			
4000 p/s	54	63	76				
5000 p/s	67	80	90				
	Max Packet / Sec		7000		**Max bits / Sec***		116Mbps

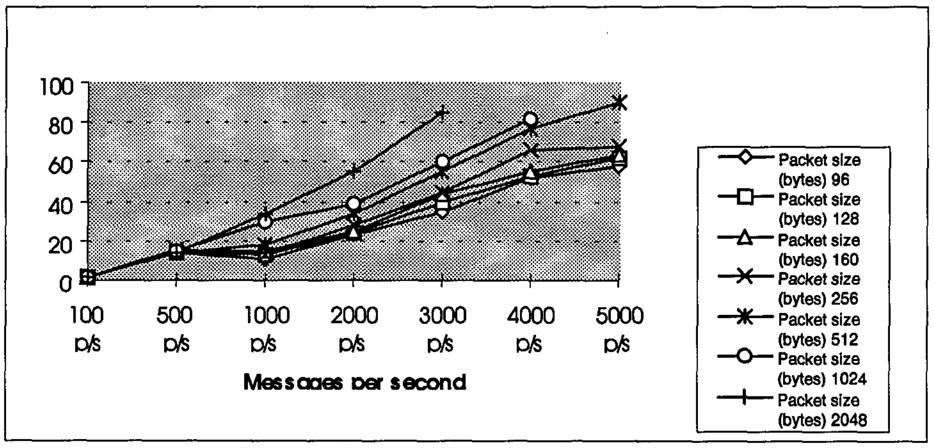

Figure 4-12: CPU utilization, Indigo2 ATM-IP

Table 4-11: CPU Utilization, Indigo2 ATM-AAL 5

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	17	17	19	18	19	25	29
500 p/s	20	22	21	22	22	23	26
1000 p/s	24	24	25	26	26	29	36
2000 p/s	32	32	31	33	38	42	53
3000 p/s	37	38	37	41	45	53	69
4000 p/s	44	43	43	46	53	67	66
5000 p/s	51	50	51	55	67	80	99
	Max Packet / Sec		4800		**Max bits / Sec**		143Mbps

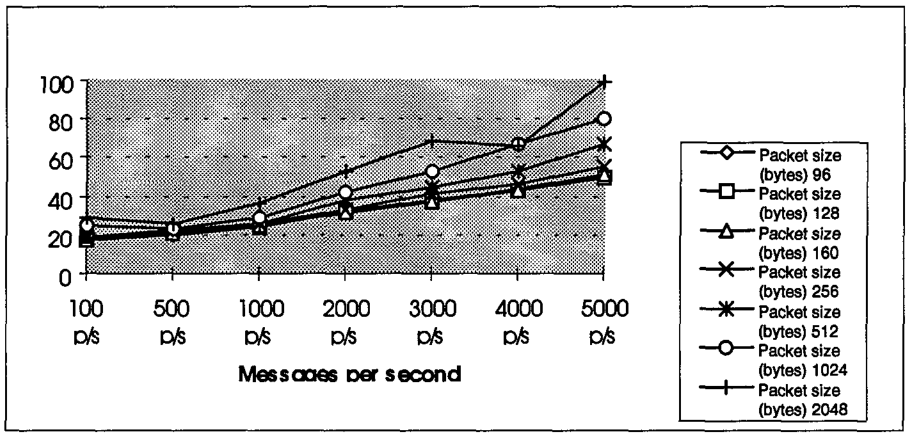

Figure 4-13: CPU utilization, Indigo2 ATM-AAL5

4.3.1.6 Other tests conducted

Two additional tests used a quad processor Onyx2 and an FDDI-based Sparc20. The Onyx2 proved to be the more efficient host on the majority of the tests. The four advanced R10000 processors provided increased processing capabilities relative to the other hosts. It is also by far the most costly host in the testing lineup. The Onyx2's performance is compared to the quad R4000 Onyx host. As the Tables 4-11 and 4-12 show, the Onyx2 is more efficient at the lower data rates but is almost the same as the Onyx above 2,000 Pk/s.

Table 4-11: CPU utilization, Onyx2 Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	96	128	160	256	512	1024	2048
100 p/s	3	3	4	4	4	4	5
500 p/s	7	7	7	7	8	9	12
1000 p/s	10	10	10	12	14	16	
2000 p/s	17	16	19	19	22		
3000 p/s	21	22	24	29			
4000 p/s	27	30	30				
5000 p/s	33	36	36				
	Max Packet / Sec		7700		Max bits / Sec*		142Mbps

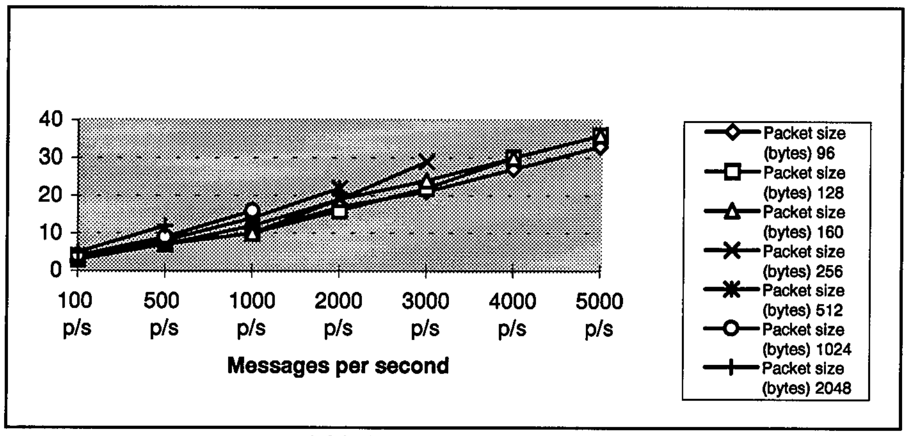

Figure 4-14: CPU utilization, Onyx2 Ethernet

Table 4-12: CPU utilization, Onyx Ethernet

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	96	128	160	256	512	1024	2048
100 p/s	4	5	5	7	9	10	12
500 p/s	8	7	12	15	19	21	25
1000 p/s	13	13	16	17	18	19	
2000 p/s	23	27	28	30	32		
3000 p/s	34	35	40	42			
4000 p/s	42	44					
5000 p/s							
	Max Packet / Sec		6800		Max bits / Sec*		151Mbps

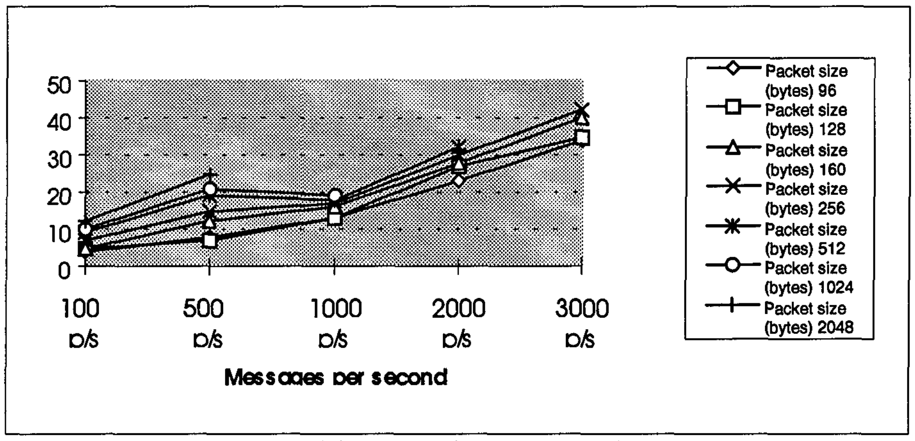

Figure 4-15: CPU utilization, Onyx Ethernet

The last test performed was on an FDDI-based Sparc20. This test reveals the packet per second limitation of the FDDI network media. This is probably due to the nature of token passing, where each host must 'touch' the packet as it passes around the ring. As more hosts enter the FDDI ring, the packet is received and transmitted more often. The FDDI network quickly becomes the bottleneck in the packet-limited environment even with its 100Mbps bandwidth. This shows that a higher bandwidth network does not always yield an increase in performance. The only way for FDDI to reach the 100Mbps data rate is to have a very large MTU, much like ATM. The FDDI MTU is 4500 bytes per frame, three times that of Ethernet. Table 4-13 shows results for the Sparc20.

Table 4-13: CPU utilization, Sparc20 FDDI

% CPU utilization rate as a function of Packet size (bytes)

Pkts/Sec	**96**	**128**	**160**	**256**	**512**	**1024**	**2048**
100 p/s	3	3	4	4	4	4	4
500 p/s	8	9	8	9	9	11	14
1000 p/s	16	16	16	16	18	21	29
2000 p/s	29	29	32	32	32	39	47
3000 p/s	45	45	46	47	52	60	62
4000 p/s							
5000 p/s							
	Max Packet / Sec	3000			**Max bits / Sec***	126Mbps	

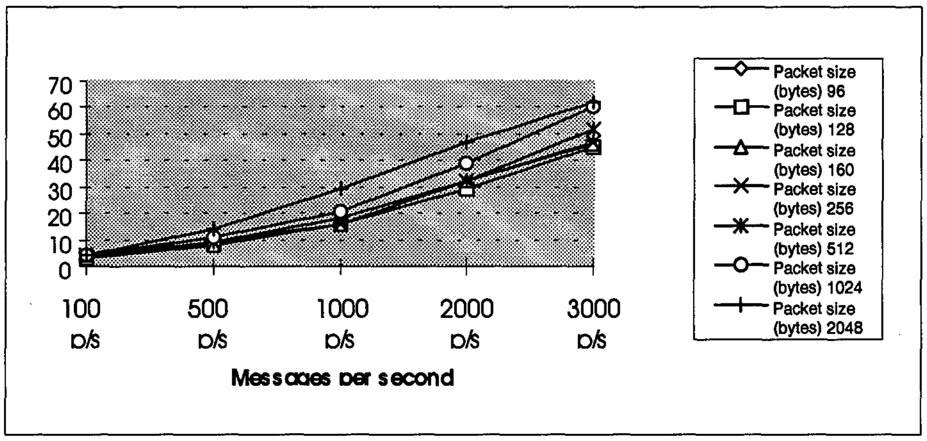

Figure 4-15: CPU utilization, Sparc20 FDDI

4.3.2 Packet test analysis

The Ethernet media has a maximum transmission unit (MTU) size of 1500 Bytes. The ATM network had an MTU of 9188 Bytes, which gives it an advantage when it comes to handling large volumes of data. This advantage of ATM is evident in the maximum data transfer rates of the two media.

A graphical CPU meter confirmed various data points in the table. The CPU meter agreed with the netperf data to within 5% for all tests verified using the CPU meter. At the higher levels of CPU utilization, the meter reported that more than 50% of the actual CPU's usage handled network interrupts, not data. This supports the assumption that per packet processing is more CPU intensive than per byte processing. Even on the quad processor, only one CPU handled network traffic. When this one CPU reached saturation, data throughput collapsed, putting an artificial limit on CPU utilization. While adding additional processors will free up CPU time for other tasks, it can not increase the amount of data or number of packets that the native architecture can support. This became evident for the Onyx processor running ATM. As the CPU responsible for handling network traffic reached 100% utilization, the throughput collapsed, even though overall CPU usage was around 40%.

As the packet size increased, the rise in CPU utilization was not as pronounced as when the packet rate increased. All of the SGI workstations using the IP protocol had between 9% and 21% processor usage for the 500 Pk/s, 1024 Bytes per packet case. The average was 15%, showing a 6% deviation across all SGI hosts for this size / rate combination. At the higher packet rate (4,000 Pk/s) and lower data size (128 Bytes) end of the spectrum, the utilization rates were from 30% to over 60% for the same SGI hosts. These utilization's are for the exact same data rate (512Kbytes / sec) as the previous case.

A general conclusion is that each host has a limit of about 5,000 to 6,000 Pk/s, regardless of media, host processing speed or operating system. The case of the 686 processor and the Onyx2 shows that future hardware and operating system optimizations can go beyond this limitation, but only with extremely high CPU usage.

4.4 Converter process results

Results from the converter process tests and simulations show that as data rates increase, the converter processes can keep the CPU utilization rates of the hosts relatively low. This architecture also breaks the 5,000 message per second barrier, sending as many as 10,000 messages per second during the highest volume loads with the simulation host processing this data at roughly 13% CPU utilization. There is a 'break-even' point between the converted and non-converted network of 10% CPU utilization or about 1,200 messages per second. The break-even point comes about because all incoming message traffic, regardless of message address, is sent to the simulation host for processing at 200 Pk/s.

One other benefit of the converter process is that the simulation host CPU load remained relatively constant. Even if the simulation host does not average 1,200 messages per second, it may have periods where the loading becomes significant enough to cause performance degradation. In this case, the converter could prevent these performance lags. In a real-time interactive simulation, these performance lags could be severe enough to jeopardize the simulation's usefulness [2]. The final benefits, message priority and flow control, can 'artificially' limit the amount of traffic a host might get. Even though the network media, host interface and operating system could handle these new message bundles, the simulation might not be able to handle the tremendous volume. In light of this, the receiving host could establish its own internal saturation levels. If the message traffic exceeds this saturation level, the host interface process would start rejecting non-priority traffic.

There were five topologies for the simulations; the baseline, the switched 100Mbps baseline, the baseline with converter, the 100/10Mbps converter and the 100/100Mbps converter.

4.4.1 Baseline

The baseline performed as predicted and matched the real network test-bed's results using the osview performance meter. The traffic generated for this test pushed each host uniformly. While this does not conform to expected traffic, it does allow the network to run at peak efficiency. The actual traffic makeup is still unknown at this time [18], but the converter process was compared with the most highly optimized conventional network traffic. The performance data seen in Tables 4-14 and 4-15 reflect that the Ethernet link is the bottleneck for high-speed data. As link utilization goes beyond 70% (3,000 messages per second), delays begin to increase exponentially. Table 4-15 illustrates these measures. The first number (250 or 450) is the number of messages being generated by the host and the second number (1,000 to 7,000) indicates the number of incoming messages from the Wide Area Network.

Table 4-14: Baseline link utilization

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
host-link %	53.9	69.5	71.5	71.9	82.3	93.2	89.3

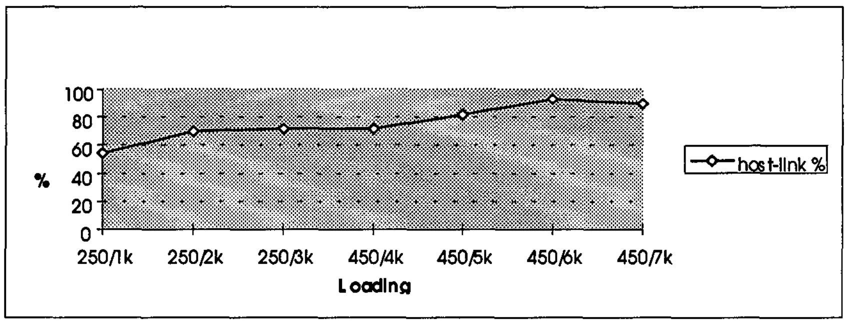

Figure 4-16: Baseline link utilization

Table 4-15: Baseline CPU use, delay

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
Host %	33.7	49.2	54.1	58.4	64.7	91.4	82.3
Msg delay (ms)	1.5	95	220	360	515	1345	3000+

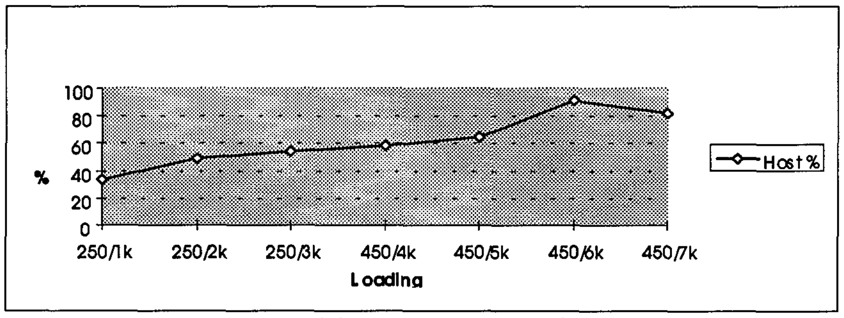

Figure 4-17:Baseline CPU use, delay

4.4.2 Switch 100Mbps Ethernet baseline

A common network upgrade is to change the media to 100Mbps switched Ethernet. This is the new 'standard' Ethernet media being shipped on all new computer equipment. At first glance, it seems that increasing the media speed would eliminate the bottleneck, and therefore greatly increase network performance. Unfortunately the CPU's themselves quickly become the bottlenecks as they reach 100% utilization.

While the message per second rate does approach the magic 5,000 mark it does so at a cost of over 90% CPU usage. For a single CPU system, this means the simulation has virtually stopped dead in its tracks. If the hosts were multiple CPU systems, the hosts would still be capable of processing the simulation, but could not handle additional network data. At this point, message delay quickly increases because the CPU cannot process the data as fast as it is coming in from the network. Tables 4-16 and 4-17 illustrate this case.

Table 4-16: 100Mbps link use

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
Host-Link %	5.4	7.1	8.6	11.2	12.7	13.6	14.9

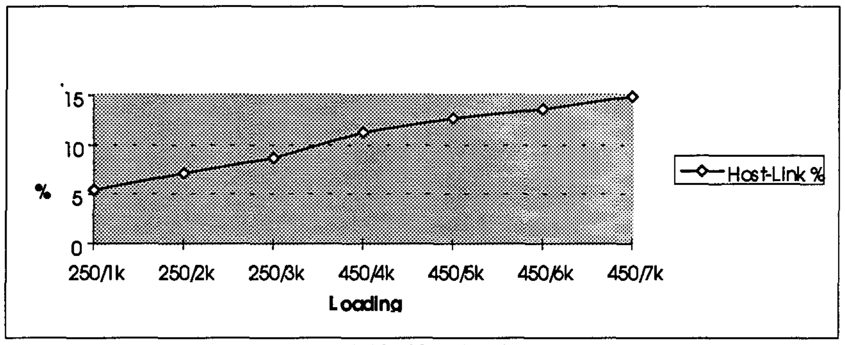

Figure 4-18: 100Mbps link use

Table 4-17: 100Mbps CPU use, delay

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
Host %	33.8	49.3	65.1	84.7	99.5	100	100
Msg delay (ms)	0.6	0.5	0.7	1.2	6.5	195	565

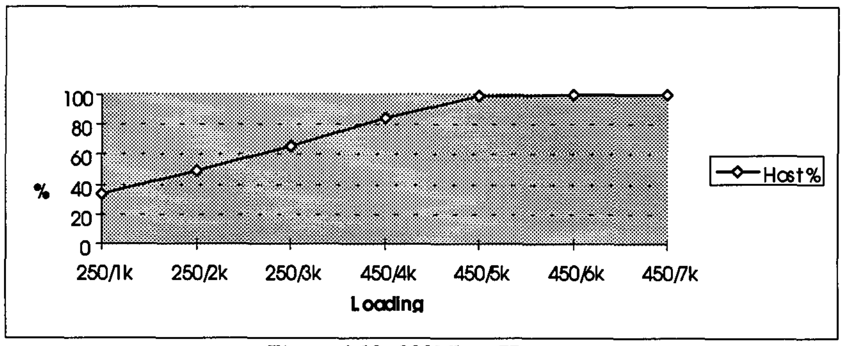

Figure 4-19: 100Mbps CPU use

4.4.3 10Mbps Converter

Unfortunately, the converter process puts too much of a data load on the local 10Mbps Ethernet. The network saturated at 1,000 messages per second with a delay of over 200milliseconds, which is beyond tolerance of the application.

4.4.4 10/100Mbps Converter

This test proved that the host CPU usage begins at about 10% and is less than 15% at over 7,000 messages per second. The two C-1 hosts demonstrate the shared loading of the incoming and outgoing data traffic as they approach 90% utilization. The extremely high utilization levels are now at the C_1 hosts instead of the simulation hosts. Delays are also below 100 milliseconds up to 6,000 Pk/s. Beyond this rate, the host 10Mbps Ethernet link begins to saturate and delays the messages. Tables 4-18 and 4-19 illustrate these tests.

Table 4-18: 100/10Mbps Link use

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
C1-Router	4.4	6.2	7.9	10.2	12.5	13.7	15.3
C1-C2	9.1	10.2	11.4	9.2	11.3	12.4	13.8
C2-Host	52.4	51.8	51.7	61.4	75.2	66.7	67.3

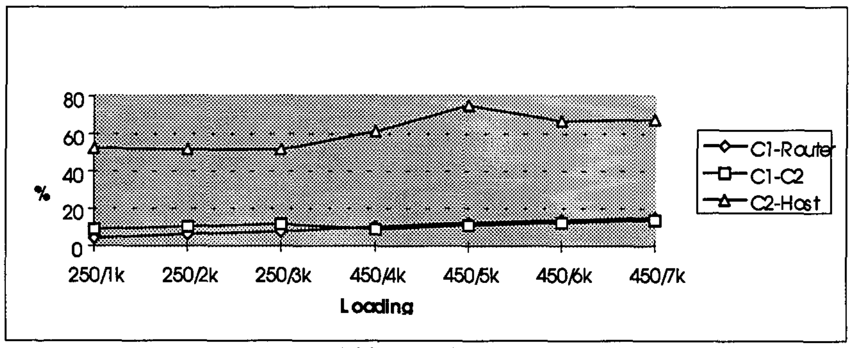

Figure 4-20: 100/10Mbps link use

Table 4-19: 100/10Mbps CPU use, delay

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
C1	32.6	40.9	49.1	62.5	75.1	81.8	91.5
C2	39.2	39.3	40.9	41.7	48.8	47.4	49.7
Host	9.9	9.9	9.8	11.3	12.4	11.5	11.6
Msg delay (ms)	12.2	18.5	23.1	45	65	111	198

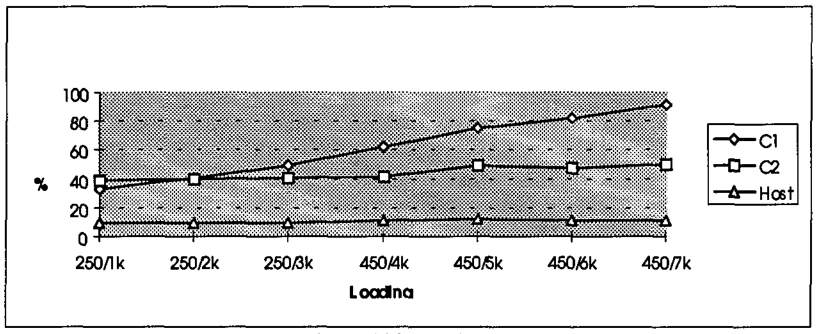

Figure 4-21: 100/10Mbps CPU use

The 100/100Mbps topology performed very much like the 100/10Mbps up to 5,000 messages per second. The most noticeable difference was the host link utilization and message delays. The message delays remain below 100 milliseconds, due to the higher bandwidth of the host link. Even at the highest traffic level the delays are still well below 100 milliseconds. The C_1 processors were approaching maximum CPU usability at the highest traffic point, so anything beyond this level would require additional C_1 hosts. The C_1 hosts saturate due to the high number (over 5,000) of incoming and outgoing IPmc packets. The C_2 processor was operating at a maximum of about 55% usage, and the data links were below 25% indicating that the system could handle even more data. Since a three or four C_1 converter based system could not be implemented in the test lab, only the two C1 converter based system is tested in the modeling environment. Additionally, the C_1 converters model the 586-based system. The C_1 converters would handle even higher data rates using the 686 system as a model.

Table 4-20: 100/100Mbps link use

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
C1-Router	4.2	6.4	7.9	8.5	12.7	14.2	16.1
C1-C2	9.1	10.3	11.5	13.5	18.4	19.2	20.9
C2-Host	5.2	5.3	5.3	6.5	7.9	7.9	8.1

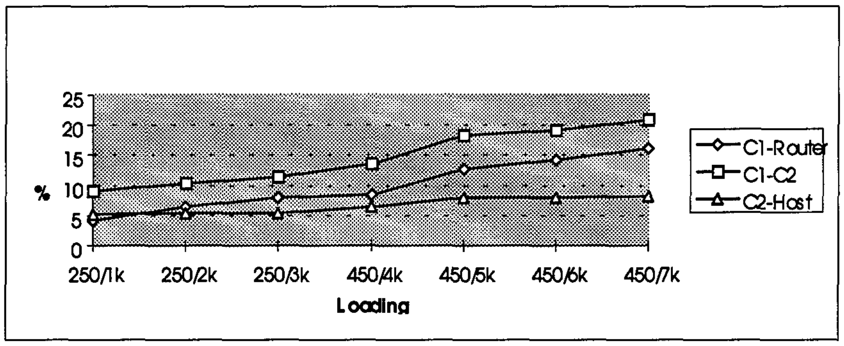

Figure 4-22: 100/100Mbps link use

Table 4-21:100/100Mbps CPU use, delay

From: Host/Router	250/1k	250/2k	250/3k	450/4k	450/5k	450/6k	450/7k
C1	38.6	40.5	48.5	55.5	75.5	83.7	94.5
C2	38.9	39.4	40.9	45.5	49.6	50.9	53.1
Host	9.9	10	10.1	11.5	12.5	12.5	12.7
Msg delay (ms)	12.1	12.2	12.5	13.1	14.3	18.4	22.5

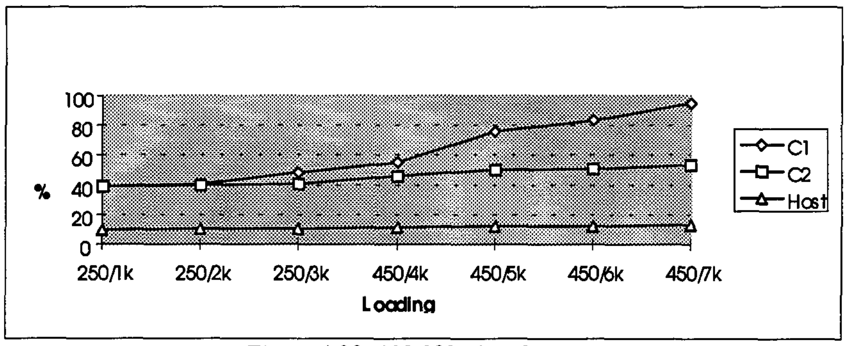

Figure 4-23: 100/100Mbps CPU use

4.5 Results summary

The tests confirmed that host processors could handle larger packets at a lower rate better than they can handle smaller packets at a higher rate. The tests also confirmed that 5,000 Pk/s is a maximum for virtually every host and on all networking media. While the converter process was not run in an actual HLA simulation, a graphical simulation with the same data traffic as in the COMNET III models was executed. In addition, an interactive simulation on an Indigo2 with a 'frames-per-second' meter as the performance indicator was conducted. The simulation could process 4.2 frames-per-second at idle and dropped to 1.8 frames-per-second at 4,000 messages-per-second. The same simulation using the converter process provided 3.8 frames-per-second at the same 4,000 message-per-second load. This confirmed the 10% CPU utilization level indicated by the model.

These results indicate that there is room for optimization within the data delivery structure of the DIS / HLA infrastructure. With minimal retrofitting, namely the addition of the address field within the message itself, the converter process provides for a scaleable, inexpensive option to implement these optimizations. Shifting the network intensive loading off the simulation hosts will allow for higher message throughput, eliminate simulation 'brown-outs' and provide for a customizable priority based message handling system.

5. Conclusions

5.1 Summary of Research / Proposal

This thesis researched several areas of a computer host's performance operating in a network environment. These areas included IP multicast efficiency, host CPU utilization under various data loads, and maximum data throughput. While IP multicast traffic proved to have an optimal range of CPU performance at lower group counts, the performance degradation had a gradual increase in relation to the group count. In short, IP multicast would not be a true 'show stopper' for DIS and HLA performance requirements. Data throughput based upon the one-message-per-packet paradigm did have dramatic limitations, especially at the higher data rates. Based upon this limitation to data throughput, an implementation was proposed that would combine, or bundle, multiple messages into a packet for efficient data delivery.

5.2 Contributions

This 'bundle-converter' process provides the benefits of lower CPU utilization and data prioritization over conventional data delivery methods. The converter does have a 'break-even' point of 1,500 messages-per-second, but this level includes both transmitted and received message traffic. Above the 1,500 message-per-second level, the converter process reduces CPU utilization on the simulation host to a range of 10 to 15 percent. As the message-per-second load increases beyond 5,000 messages-per-second, the converter process benefit of prioritization becomes evident. In a conventional network delivery system, if the data traffic bursts beyond what the host or network can handle, there is no way to determine which message traffic will be lost. The converter process tags each message with a priority status. If network or host congestion occur, the converter process will drop non-priority traffic to reduce the congestion. This ensures that priority traffic has the best chance for delivery.

5.3 Future Work

While the converter process does provide advantages over conventional networking, there are several areas where further networking performance research is required. Additional networking media should be fully researched in terms of performance criteria, the flow control scheme should be optimized according to traffic patterns and a Wide Area Network (WAN) extension to the converter process should be explored.

5.3.1 Additional Network Media Research

The networking media at AFIT limited the research to UDP/IP over OC-3 ATM, 10Base Ethernet and FDDI. Networks such as 100Base Ethernet, Token Ring, Token Bus and other media should be researched for the same per-packet limitations as seen here. If there is a networking media/protocol combination that significantly reduces the per-packet processing, it might be a good alternative to the hardware based bundle-converter system presented here.

5.3.2 Converter Flow Control Optimization

The flow control in the bundle-converter system is a simple on-off mechanism based upon media utilization. When data traffic for HLA becomes available, an optimized flow control scheme can be developed that would incorporate media utilization, data distributions and host signaling. This combination of flow-control parameters would ensure optimum data flow to the host under all data loading conditions.

5.3.3 WAN Extensions

While the converter process is intended for local network use, a possible wide area adaptation could be developed. The WAN extension could utilize a 'flow-tag' process allowing several C_2 converters to be joined together as a superset of converters. This would allow the bundles to be transmitted to each LAN directly, reducing the per-packet processing and bandwidth overhead on the WAN infrastructure.

5.4 Conclusions and Recommendations

The converter process provides for optimized data communications under moderate to heavy data loading within the DIS/HLA arena, reducing 80% CPU usage at 4,000 messages-per-second to only 13% for the same data load. The cost of additional hardware will restrict the initial feasibility of the converter process to high traffic sections within the HLA arena. The probability of data loss and performance degradation is based upon the traffic load experienced by the host. As this number of simulated entities and the fidelity of the simulations increase, there will be a greater chance of lost data and reduced system performance. Comparing the performance of a conventional HLA system to a bundle-converter based HLA system will give a definite cost performance analysis. Using the probability of data loss and performance degradation, a user can determine if and when to deploy the bundle-converter system.

Appendix A: Implementation Data

This appendix includes details for implementing the bundle converter within an SGI environment. Several items require specific mention, namely the *sproc()* function, memory use and the *usleep()* command when used in a graphics environment.

The *sproc()* function generates a stand alone child process that can share memory with the parent and sibling processes. While the *sproc()* function is truly essential in this process, it is specific to the SGI machines. Other functions provide similar functionality, such as *clone()* with Linux and *threads()* with Posix C, but most have machine dependent structures. For this reason, only SGI and Linux code are presented in detail. Other system implementations can be used, just check the specifics for the code syntax.

Memory management on an SGI machine not as bullet proof as originally anticipated, especially when graphics processes are used. I initially went with a static memory allocation, but any graphics initialization routine caused these static pointers to occasionally become invalid. This problem cleared up when I went to a *malloc()* based memory assignment. Still, care should be taken to 'refresh' these pointers when using long term memory pointers within an SGI processor.

Millisecond timing within the converter process is critical, so the Unix *usleep()* function is used. When multiple processes are running, as in our case, occasionally one of the processes would go comatose, calling usleep, but never receiving a wake-up call. The exact cause is not certain, but a fix was to refrain from running graphics programs on the SGI host running that process. The problem appears only for the C_2 and C_1 processes, which should be running on Linux boxes anyway, so a detailed fix action is not required.

Configuring a Linux host to use IP multicast in the bundle converter system has a few special steps, as follows:

A. Install the Linux version 4.1 or higher system from CD-ROM with all possible components. This will require about 400Meg of hard drive space.

B. Change the following line in /usr/include/linux/socket.h from 20 to 2000 i.e.

```
#define IP_MAX_MEMBERSHIPS          2000
```

C. Run the 'make menuconfig' option as outlined in the installation manual and chose the 'advanced / developer options'. Scroll to the network section and chose IP multicast enabled.

D. Recompile the kernel and store it as the boot image file. Reboot.

These steps assume some familiarity with Linux, but any good reference manual will be able to fill in the specific details for your version.

Appendix B: C Code

C_1 Converter Process

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <netdb.h>
#include <stdio.h>

#include <linux/unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#define STACKSIZE 16384

#define CSIGNAL        0x000000ff    /* signal mask at exit */
#define CLONE_VM       0x00000100    /* set if VM shared */
#define CLONE_FS       0x00000200    /* set if fs  shared */
#define CLONE_FILES    0x00000400    /* set if files shared */
#define CLONE_SIGHAND  0x00000800    /* set if sig.hand.shared */


#define BUFFSIZE 65536 /* maximum buffer size */
#define MBUFFSIZE 16384 /* maximum message size */
#define MAX_OUT 30 /* maximum outstanding bundle transmissions is 30 */
#define MAX_MSG 150 /* maximum number of messages per bundle...*/
#define BASE_ADDR "234.5.1.2"
#define R_SND_ADDR "234.6.1.1"
#define R_RCV_ADDR "234.5.1.1"
#define C2_SND_ADDR "234.4.1.1"
#define C2_RCV_ADDR "234.4.1.10"
#define R_CTL_ADDR "234.10.1.1"
#define C1_MASK 49153 /* individual mask for the C1 Converter + 49152 */
```

```
#define GRP_TO_ADDR 32 /* number of groups per IPmc address, must be */
#define R_TTL_VALUE 1                  /* a number 2^l */
#define C2_TTL_VALUE 1
#define CONGEST_ALLOWED 7 /* allow x - C2_CONGEST msg when congested */
#define R_PORT 3456
#define C2_RCV_PORT 3457
#define C2_SND_PORT 3450
#define TRUE 1
#define FALSE 0
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

/* build a structure here so that we can access it like an array...*/
struct mr_buffer { /* message recieve buffer */
   char BUF[BUFFSIZE];
   u_short END[MAX_MSG];
} MSG_REC[MAX_OUT]; /* allow for outstanding message transmissions */

struct end_buffer {
   u_short BUF[MAX_MSG];
} END[MAX_OUT]; /*establish set of end pointers, 2 char (short int..) */

struct br_buffer { /* bundle recieve buffer */
   char BUF[BUFFSIZE];
   u_short LEN;
} BDL_REC[MAX_OUT]; /* allow for outstanding bundle transmissions */

struct sockaddr_in Router, C2send, C2recv; /*socket stuctures */
struct ip_mreq stMreq;

/* set up globals (ALL CAPS) for sproc processes */
u_char CUR_RCV_BBUF, BDL_SND_C, MSG_RCV_C, MSG_SND_C, EXCEED;
u_short MSG_RCV_L, R_CONGEST, BBUF_LOCK, MSG_CNT;
int MSG_SEND_PTR, MSG_SENT_PTR, MSG_IN, MSG_OUT, C2_IN, C2_OUT;
int TIME_DELAY, C2_CONGEST;
int R_SOCK, C2_SND_SOCK, C2_RCV_SOCK;
char MSG_REC_BUF[MBUFFSIZE]; /* a scratch pad buffer for recv_msg */
char CTMP;
char* MSG_BUFFPTR;
int addr_size = sizeof(struct sockaddr_in); /* socket stuff again */

void join_addr(int l_addr_offset);

void drop_addr(int l_addr_offset);

void send_bdl(u_char l_buf,u_char l_rcv_c, u_short l_rcv_l);

void recv_bdl(void * data);

void send_msg(void * data);

void recv_msg(void * data);

void init_prc_c1(int argc,char *argv[]);

void status(void * data);

int start_thread(void (*fn)(void *), void *data)
{
        long retval;
        void **newstack;

        /*
         * allocate new stack for subthread
         */
        newstack = (void **) malloc(STACKSIZE);
        if (!newstack)
                return -1;

        /*
         * Set up the stack for child function, put the (void *)
         * argument on the stack.
         */
        newstack = (void **) (STACKSIZE + (char *) newstack);
        *--newstack = data;

        /*
         * Do clone() system call. We need to do the low-level stuff
         * entirely in assembly as we're returning with a different
         * stack in the child process, we couldn't otherwise guarantee
         * that the program doesn't use the old stack incorrectly.
         *
         * Parameters to clone() system call:
         *      %eax - __NR_clone, clone system call number
         *      %ebx - clone_flags, bitmap of cloned data
         *      %ecx - new stack pointer for cloned child
         *
         * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
         * CLONE_SIGHAND which shares as much as possible between parent
         * and child. (We or in the signal to be sent on child termination
         * into clone_flags: SIGCHLD makes the cloned process work like
         * a "normal" unix child process)
         *
         * The clone() system call returns (in %eax) the pid of the newly
         * cloned process to the parent, and 0 to the cloned process. If
         * an error occurs, the return value will be the negative errno.
         *
         * In the child process, do a "jsr" to the requested function
         * and then do a "exit()" system call, terminate the child.
         */
        __asm__ __volatile__(
                "int $0x80\n\t"                 /* Linux/i386 system call */
                "testl %0,%0\n\t"       /* check return value */
                "jne 1f\n\t"            /* jump if parent */
                "call *%3\n\t"                  /* start subthread function */
                "movl %2,%0\n\t"
                "int $0x80\n"                   /* exit system call: */
                "1:\t"
                :"=a" (retval)
                :"0" (__NR_clone),"i" (__NR_exit),
                 "r" (fn),
                 "b" (CLONE_VM | CLONE_FS | CLONE_FILES |
                   CLONE_SIGHAND | SIGCHLD),"c" (newstack));

        if (retval < 0) {
                errno = -retval;
                retval = -1;
        }
        return retval;
}


void cloned_process_starts_here(void * data)
{
}

void main(argc, argv)
```

```
    int argc;
    char *argv[];
{ /* begin of main */
   int rm_pid, rb_pid, send_pid, x; /* pid variables */
   u_short l_msg_rcv_l; /*local message recieve length */
   struct timeval ta, tb;
   u_char l_buf, l_msg_rcv_c; /* local buff & msg recieve count */
   init_prc_c1(argc, argv); /* intialize the convert_1 process */
/* sproc our recieve and send processes */


   rm_pid = start_thread(status, (void *) x);
   if (rm_pid < 0) {
            perror("start_thread");
            exit(1);
   }
   rm_pid = start_thread(recv_msg, (void *) x);
   if (rm_pid < 0) {
            perror("start_thread");
            exit(1);
   }
   rb_pid = start_thread(recv_bdl, (void *) x);
   if (rb_pid < 0) {
            perror("start_thread");
            exit(1);
   }
   send_pid = start_thread(send_msg, (void *) x);
   if (send_pid < 0) {
            perror("start_thread");
            exit(1);
   }


   while(TRUE) { /* loop for ever */
            usleep(TIME_DELAY); /* wait between bundle transmissions */
            gettimeofday(&ta,0);  /* get our reference start time */
            if (MSG_RCV_C != 0) { /* if we get any messages, process now*/
               l_buf = CUR_RCV_BBUF; /* set up for bundle trans */
               while(BBUF_LOCK){
               }
               CUR_RCV_BBUF++; /* set pointer to new buf */
               if (CUR_RCV_BBUF == MAX_OUT) CUR_RCV_BBUF = 0;
               /*          printf("send bundle %d\n",CUR_TRN_BBUF); */
               l_msg_rcv_c = MSG_RCV_C; /* we might recieve a message*
                                                        * while sending the current */
               l_msg_rcv_l = MSG_RCV_L; /*bundle, so do this now...*/
               MSG_RCV_C = 0;
               MSG_RCV_L = 2;
               send_bdl(l_buf,l_msg_rcv_c, l_msg_rcv_l);
               if (TIME_DELAY > 3500) { /* light network, send all...*/
                        C2_CONGEST--;
                        if (C2_CONGEST < 0) C2_CONGEST = 0;
                        /* min congest is zero */
               }
               else if(TIME_DELAY < 1000) {
                        /* getting heavy, may need drop a few */
                        C2_CONGEST++;
                        if (C2_CONGEST > 3) C2_CONGEST = 3;
                        /* max congest is 3 */
               }
            }
            gettimeofday(&tb,0);
            TIME_DELAY = 5000 - tb.tv_usec - ta.tv_usec;
            if (TIME_DELAY > 5000) TIME_DELAY = 5000; /* longest delay*/
            if (TIME_DELAY < 1) TIME_DELAY = 1; /* shortest delay */
   }
} /*end of main proc*/

void join_addr(int l_addr_offset) {
   /* join the multicast address */
   int iRet;
   /* set join address to our 'BASE' multicast address
    *plus the offset value */
   stMreq.imr_multiaddr.s_addr =
     inet_addr(BASE_ADDR)+
     16777216*u_addr
     +65536*l_addr; /* adjust for 'inverted byte order' */
   stMreq.imr_interface.s_addr = INADDR_ANY;
   iRet = setsockopt(R_SOCK,IPPROTO_IP,IP_ADD_MEMBERSHIP,
                        (char *)&stMreq,sizeof(stMreq));
   /*actualy send the data here! */
   printf("Joined group %d\n",addr_offset);
   if (iRet == -1) {
            /* did it work? if no then print error message...*/
            perror("IP_ADD_MEMBERSHIP failed:");
   }
}

void drop_addr(int l_addr_offset) {
   /*drop the multicast address */
   int iRet;
   /* set join address to our 'BASE' multicast address
    * plus the offset value */
   stMreq.imr_multiaddr.s_addr =
     inet_addr(BASE_ADDR)+
     16777216*u_addr
     +65536*l_addr; /* adjust for 'inverted byte order' */
   stMreq.imr_interface.s_addr = INADDR_ANY;
   iRet = setsockopt(R_SOCK,IPPROTO_IP,IP_DROP_MEMBERSHIP,
                     (char *)&stMreq,sizeof(stMreq));
   if (iRet == -1) {
/* did it work? if no then print error message...*/
         perror("IP_DROP failed");
   }
}

void send_bdl(u_char l_buf, u_char l_rcv_c, u_short l_rcv_l) {
   int iRet;
   u_short *read_ptr;
   char *write_ptr;
   /*dump the entire 'END.BUF' string into the buffer, then truncate...*/
   read_ptr = &MSG_REC[l_buf].END[0]; /* read from end point buffer */
   write_ptr = &MSG_REC[l_buf].BUF[l_rcv_l]; /* write to end of buf*/
   while(BBUF_LOCK) {
   }
   memcpy(write_ptr, read_ptr, l_rcv_c*2);
   /* now end the bundle with the message count */
   sprintf((MSG_REC[l_buf].BUF+l_rcv_l+2*l_rcv_c), "%c", l_rcv_c);
   iRet = sendto(C2_SND_SOCK,MSG_REC[l_buf].BUF,
                 (l_rcv_l+2*l_rcv_c+1),0,
                 (struct sockaddr*)&C2send,addr_size);
   MSG_CNT = 0; /* reset message statistics */
   if (iRet == -1) {
         printf("C2 SND sock send failed %d\n",errno);
   }
   C2_OUT+=l_rcv_c;
}

void recv_bdl(void * data) {
   u_short iRet;
   int addr_size = sizeof(struct sockaddr_in);
   while(TRUE) { /*loop until program is exited*/
         iRet = recvfrom(C2_RCV_SOCK,BDL_REC[MSG_SEND_PTR].BUF,
                         BUFFSIZE,0,(struct sockaddr*)&C2recv,
                         &addr_size);
         C2_IN+=BDL_REC[MSG_SEND_PTR].BUF[iRet - 1];
         BDL_REC[MSG_SEND_PTR].LEN = iRet;
         MSG_SEND_PTR++;
         if (MSG_SEND_PTR == MAX_OUT) MSG_SEND_PTR = 0;
         if (MSG_SEND_PTR == MSG_SENT_PTR) MSG_SENT_PTR++;
         /* the above code will 'jump' MSG_SENT_PTR up
          * to 'over write' old data */
   }
}

void send_msg(void * data) {
   int iRet;
   u_short start, end, l_len, mc_addr, addr,len;
   u_char l_cnt, x, l_buf;
   char *read_ptr;
   u_short *write_ptr;
   int addr_size = sizeof(struct sockaddr_in);
   while(TRUE) {
         if (MSG_SENT_PTR == MSG_SEND_PTR) usleep(1000);
         else {
           start = 0;
           l_buf = MSG_SENT_PTR;
           MSG_SENT_PTR++;
           if (MSG_SENT_PTR == MAX_OUT) MSG_SENT_PTR = 0;
           l_len = BDL_REC[l_buf].LEN;
           l_cnt = BDL_REC[l_buf].BUF[l_len-1];
           read_ptr = &BDL_REC[l_buf].BUF[l_len-2*l_cnt-1];
           write_ptr = &END[l_buf].BUF[0];
           memcpy(write_ptr, read_ptr, 2*l_cnt);
           /*read number of msgs from end of buffer */
           for (x=0; x < l_cnt;x++) {
                 addr = BDL_REC[l_buf].BUF[start]*256 +
                     BDL_REC[l_buf].BUF[start+1];
                 mc_addr = addr / GRP_TO_ADDR;
                 /* divide by some number, bring down to IP mc address */
                 end = END[l_buf].BUF[x];
                 /* offset from end for pointer */
                 len = end - start;
                 if (len == 22) { /* control message */
                   switch(BDL_REC[l_buf].BUF[start+2]) {
                   case 1 : join_addr(mc_addr);
                          break;
                   case 2 : drop_addr(mc_addr);
                          break;
                   case 3 : C2_CONGEST = 3;
```

```
                        break;
                case 4 : C2_CONGEST = 4;
                        break;
                case 5 : C2_CONGEST = 5;
                        break;
                case 6 : C2_CONGEST = 6;
                        break;
                default : printf("Control message error\n");
                }
            }
            else {
        Router.sin_addr.s_addr = inet_addr(R_SND_ADDR)+mc_addr;
                MSG_BUFFPTR = &BDL_REC[l_buf].BUF[start];
                MSG_OUT++;
                /*uncomment for prod code */
                /* iRet = sendto(R_SOCK,MSG_BUFFPTR,len,0,
                        (struct sockaddr*)&Router,addr_size);
                if (i Ret == SOCKET_ERROR) {
                    if  (errno == 132) {
                        while (iRet == SOCKET_ERROR) {
                            usleep(100);
                            i Ret = sendto(R_SOCK,MSG_BUFFPTR,
                                        len,0,
                            (struct s ockaddr*)&Router,addr_size);
                        }
                    }
                    else
    printf("R_sendto() failed, Error: %dlen = %d \n", errno,len);
                } */
            }
            start = end;
        }
        } /*end else */
    }
} /* end proc send_msg */


void recv_msg(void * data) {
    int iRet, cur_pointer;
    int addr_size = sizeof(struct sockaddr_in);
    u_short l_addr;
    while(TRUE) { /* loop forever... */
            /* must recieve into temp buffer then write, since recvfrom */
            /* is a blocking function and CUR_REC_BBUF may change between */
```

```
        /* call and return */
        iRet = recvfrom(R_SOCK,MSG_REC_BUF,MBUFFSIZE,0,
                        (struct sockaddr*)&Router,&addr_size);
        if (iRet < 0) {
          printf ("recv failed, Error: %d\n", iRet);
        }
        memcpy(&l_addr,&MSG_REC_BUF,2);
        if (((C2_CONGEST<4)||(l_addr&1))||
          ((!EXCEED)&&(C2_CONGEST<6))) {
          BBUF_LOCK = TRUE;
          memcpy(MSG_REC[CUR_RCV_BBUF].BUF +
                  MSG_RCV_L,MSG_REC_BUF,iRet);
          BBUF_LOCK = FALSE;
          MSG_RCV_L+=iRet;
          MSG_REC[CUR_RCV_BBUF].END[MSG_RCV_C] = MSG_RCV_L;
          MSG_RCV_C++;
          MSG_IN++;
          if (C2_CONGEST>3) {
                MSG_CNT++;
                if (MSG_CNT > CONGEST_ALLOWED - C2_CONGEST)
                  EXCEED = TRUE;
          }
        }
    }
}


void init_prc_c1(int argc, char *argv[])
{ /*begin init_prc_c1 */
   int iTmp, iRet, x;
   u_char cTmp;
   u_short c1_mask = C1_MASK;
   struct hostent *hp, *gethostbyname();
   /* Start setting up for our Router based socket...*/
   TIME_DELAY = 5000; /* start with a 5 mil_second delay...*/
   MSG_RCV_C = 0;
   MSG_RCV_L = 2;
   for(x=0;x<MAX_OUT;x++) {
          memcpy(&MSG_REC[x].BUF[0],&c1_mask,2);
   } /* prep all buffers with sending mask id...*/
   MSG_SENT_PTR = 0;
   MSG_SEND_PTR = 0;
   R_SOCK = socket(AF_INET,SOCK_DGRAM,0);
   if (R_SOCK == INVALID_SOCKET) {
          printf ("R_socket() failed, Err: %d\n", -1);
        exit(1);
    }
    /* avoid EADDRINUSE error on bind() */
    iTmp = TRUE;
    iRet = setsockopt(R_SOCK,SOL_SOCKET,SO_REUSEADDR,
                    (char *)&iTmp,sizeof(iTmp));
    if (iRet == SOCKET_ERROR) {
        printf ("R_setsockopt() SO_REUSEADDR failed, Err: %d\n",-1);
    }
    /* name the socket */
    Router.sin_family     = AF_INET;
    Router.sin_addr.s_addr = htonl(INADDR_ANY);
    Router.sin_port       = htons(R_PORT);
    iRet = bind(R_SOCK, (struct sockaddr*) &Router,
                    sizeof(Router));
    if (iRet == SOCKET_ERROR) {
        printf ("R_bind() failed, Err: %d\n",-1);
    }
    /* set TTL to traverse up to multiple routers */
    cTmp = R_TTL_VALUE;
    iRet = setsockopt(R_SOCK,IPPROTO_IP,IP_MULTICAST_TTL,
                    (char *)&cTmp,sizeof(cTmp));
    if (iRet == SOCKET_ERROR) {
        perror("IP_multicast TT failed");
    }
    /* disable loopback */
    cTmp = FALSE;
    iRet = setsockopt(R_SOCK,IPPROTO_IP,IP_MULTICAST_LOOP,
                    (char *)&cTmp,sizeof(cTmp));
    if (iRet == SOCKET_ERROR) {
        perror("IP_multicast loop failed");
    }
    /* join the multicast address to send out Router Control traffic */
    stMreq.imr_multiaddr.s_addr = inet_addr(R_RCV_ADDR);
    stMreq.imr_interface.s_addr = INADDR_ANY;
    iRet = setsockopt(R_SOCK,IPPROTO_IP,IP_ADD_MEMBERSHIP,
                    (char *)&stMreq,sizeof(stMreq));
    if (iRet == -1) {
/* did it work? if no then print error message...*/
        perror("IP_multicast add member failed");
    }

    /* so far so good, now lets set up our C2 send based socket...*/

    C2_SND_SOCK = socket(AF_INET,SOCK_DGRAM,0);
    if (C2_SND_SOCK == INVALID_SOCKET) {
        printf ("C2S_socket() failed, Err: %d\n", -1);
        exit(1);
    }
    /* name the socket */

    hp = gethostbyname(argv[1]);
    if (hp == 0) {
        printf(" unknown host\n");
        exit(2);
    }
    memcpy( (char *)&C2send.sin_addr, (char*)hp->h_addr,
            hp->h_length);
    C2send.sin_family =          AF_INET;
    C2send.sin_port =   htons(C2_SND_PORT);


    /* so far so good, now lets set up our C2 RCV based socket...*/

    C2_RCV_SOCK = socket(AF_INET,SOCK_DGRAM,0);
    if (C2_RCV_SOCK == INVALID_SOCKET) {
        printf ("C2R_socket() failed, Err: %d\n", -1);
        exit(1);
    }

    C2recv.sin_family     = AF_INET;
    C2recv.sin_addr.s_addr = htonl(INADDR_ANY);
    C2recv.sin_port       = htons(C2_RCV_PORT);
    iRet = bind(C2_RCV_SOCK, (struct sockaddr*)&C2recv,
                    sizeof(C2recv));
    if (iRet == SOCKET_ERROR) {
        printf ("C2R_bind() failed, Err: %d\n",-1);
    }
}

void status(void * data) {
    while(TRUE) {
        sleep(1);
   printf("MSG_IN %d  MSG_OUT %d  C2_IN %d  C2_OUT %d MSG_RCV_C %d\n",
        MSG_IN, MSG_OUT, C2_IN, C2_OUT, MSG_RCV_C);
        MSG_IN = 0;
        MSG_OUT = 0;
        C2_IN = 0;
        C2_OUT = 0;
    }
```

C 2 Converter Process

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <unistd.h>
#include <sys/time.h>
#include <linux/unistd.h>

#define STACKSIZE 16384

#define CSIGNAL        0x000000ff     /* signal mask at exit */
#define CLONE_VM        0x00000100     /* set if VM shared */
#define CLONE_FS        0x00000200     /* set if fs shared */
#define CLONE_FILES     0x00000400     /* set if files shared */
#define CLONE_SIGHAND   0x00000800     /* set if sighand shared */


#define BUFFSIZE 65536 /* maximum buffer size */
#define MAX_OUT 10    /* maximum outstanding bundle transmissions */
#define MAX_MSG 150   /* maximum number of messages per bundle */
#define C_ADDR "234.4.1.10"
#define H_ADDR "234.3.1.1"
#define RCV_ADDR "234.4.1.1"
#define C1_HOST_A "donatello"
#define C1_HOST_B "louvre"
#define C1_HOST_C "next_hosts_name..."
#define H_HOST "rembrandt"
#define GRP_TO_ADDR 32 /* group to IPmc address conversion */
#define H_TTL_VALUE 1
#define C_TTL_VALUE 1
#define H_SND_PORT 3459
#define H_RCV_PORT 3458
#define C_RCV_PORT 3450
#define C_SND_PORT 3457
#define H_CONGEST_ALLOWED 7
```

```
#define C1_CONGEST_ALLOWED 7
#define TRUE 1
#define FALSE 0
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
#define C1_CONV 3 /* define the number of C1 converters plus 1
                   * NOTE: C1 converters are referenced starting with 1
                   * instead of zero so we must add 1!!
                   /* If a C1 converter had
                  * an index of 0 (zero) it wouldn't show up in the IPMC
                  *database!! Speaking from experience here... */

/* the following media capacities reflect the number of bytes
 * that can be sent during a 5ms interval, based on bandwidth
 * ethernet is 6000, Fast ether is 60000, ATM is 62000 etc */

#define MEDIA_CAP_H 6000  /* ethernet */
#define MEDIA_CAP_C1 6000 /* ethernet */

/* Create the threshold values for when we need to change
 * our congestion states. 80% upper and 40% lower are
 * experimental values, research will dictate optimum numbers... */

u_short THRESH_H_UPPER = .8*MEDIA_CAP_H;
u_short THRESH_H_LOWER = .4*MEDIA_CAP_H;
u_short THRESH_C1T_UPPER = .8*MEDIA_CAP_C1;
u_short THRESH_C1T_LOWER = .4*MEDIA_CAP_C1;
u_char H_CONGEST, C1_CONGEST; /* congestion state */

struct b_buffer {      /* built as struct for array ref... */
   char *BUF;
   u_short LEN;        /* length of current buffer in bytes */
   u_char CNT;         /* number of messages within buffer */
   u_short *END;       /*record where each msg ends*/
} BDL_REC[3], H_SND[MAX_OUT], C_SND[MAX_OUT][C1_CONV];

u_short BD_CAST = 65535;
u_short *MASKDB; /* mask data base structure */
u_short *IPMCDB; /* IP multicast address data base */
struct sockaddr_in H_SND_STR, C_SND_STR[C1_CONV],
            H_RCV_STR, C_RCV_STR; /*socket stuctures */
```

```
/* set up globals (ALL CAPS) for sproc processes */
u_char RCV_BBUF, CUR_SND_HBUF, CUR_SND_CBUF[C1_CONV];
u_char H_EXCEED, C1_EXCEED[C1_CONV];
int H_SND_SOCK, C_SND_SOCK[C1_CONV], H_RCV_SOCK, C_RCV_SOCK;
char CTMP;
char *MSG_BUFFPTR; /* pointer for buffer space */

/* lock value and current data volumes for host C1 total and C1s */
u_short BDL_LOCK, DATA_VOL_H, DATA_VOL_C1T;
u_short H_MSG_CNT, C1_MSG_CNT;
int DATA_VOL_C1[C1_CONV]; /* stats for each C1 converter */

/* statistics variables for status demo, remove for final code */
u_short H_IN, H_OUT, C_IN, C_OUT;

void host_xmt(void * data);

void c1_xmt(void * data);

void host_rcv(void* data);

void update_mask(u_short h_mask, u_short l_addr, u_char l_act);

void ctl_snd(u_short l_addr,u_char c1, u_char l_act);

void proc_bdl(u_char l_buf);

void proc_c_snd(u_short gr_addr, u_short l_ptr,
                u_short l_end, u_char b_p);

void proc_h_snd(u_short gr_addr, u_short l_ptr,
                u_short l_end, u_char b_p);

void init_prc_c2(void);

void status(void * data);


int start_thread(void (*fn)(void *), void *data)
{
        long retval;
        void **newstack;

        /*
         * allocate new stack for subthread
         */
        newstack = (void **) malloc(STACKSIZE);
        if (!newstack)
                return -1;

        /*
         * Set up the stack for child function, put the (void *)
         * argument on the stack.
         */
        newstack = (void **) (STACKSIZE + (char *) newstack);
        *--newstack = data;

        /*
         * Do clone() system call. We need to do the low-level stuff
         * entirely in assembly as we're returning with a different
         * stack in the child process and we couldn't otherwise guarantee
         * that the program doesn't use the old stack incorrectly.
         *
         * Parameters to clone() system call:
         *        %eax - __NR_clone, clone system call number
         *        %ebx - clone_flags, bitmap of cloned data
         *        %ecx - new stack pointer for cloned child
         *
         * In this example %ebx is CLONE_VM | CLONE_FS | CLONE_FILES |
         * CLONE_SIGHAND which shares as much as possible between parent
         * and child. (We or in the signal to be sent on child termination
         * into clone_flags: SIGCHLD makes the cloned process work like
         * a "normal" unix child process)
         *
         * The clone() system call returns (in %eax) the pid of the newly
         * cloned process to the parent, and 0 to the cloned process. If
         * an error occurs, the return value will be the negative errno.
         *
         * In the child process, we will do a "jsr" then do a
        /* "exit()" system call which will terminate the child.
         */
        __asm__ __volatile__(
                "int $0x80\n\t"             /* Linux/i386 system call */
                "testl %0,%0\n\t"      /* check return value */
                "jne 1f\n\t"           /* jump if parent */
                "call *%3\n\t"              /* start subthread function */
                "movl %2,%0\n\t"
                "int $0x80\n"               /* exit system call: */
                "1:\t"
                :"=a" (retval)
                :"0" (__NR_clone),"i" (__NR_exit),
                 "r" (fn),
                 "b" (CLONE_VM | CLONE_FS | CLONE_FILES |
                    CLONE_SIGHAND | SIGCHLD),"c" (newstack));

        if (retval < 0) {
                errno = -retval;
                retval = -1;
        }
        return retval;
}


void cloned_process_starts_here(void * data)
{
}

void main(void)
{ /* start of main */
  int s_pid, hx_pid, hr_pid, c1_xpid, x, y;
  int l_addr_size = sizeof(struct sockaddr_in);
  u_short iRet;
  u_char c = 0;
  BDL_LOCK = 0;

  BDL_REC[0].BUF = malloc(BUFFSIZE); /* this is done via malloc */
  BDL_REC[1].BUF = malloc(BUFFSIZE); /* to avoid memory conflicts */
  BDL_REC[2].BUF = malloc(BUFFSIZE); /* with graphic processes.. */
  BDL_REC[0].END = malloc(2*MAX_MSG);
  BDL_REC[1].END = malloc(2*MAX_MSG);
  BDL_REC[2].END = malloc(2*MAX_MSG);
  MASKDB = malloc(2*BUFFSIZE);
  IPMCDB = malloc(2*BUFFSIZE);

  for(x=0;x < MAX_OUT;x++) {
        H_SND[x].BUF = malloc(BUFFSIZE);
          H_SND[x].END = malloc(2*MAX_MSG);
          for(y=0;y < C1_CONV;y++) {
                C_SND[x][y].BUF = malloc(BUFFSIZE);
                C_SND[x][y].END = malloc(2*MAX_MSG);
          }
  }
  init_prc_c2();
```

```
s_pid = start_thread(status, (void *) x);
if (s_pid < 0) {
        perror("start_thread");
        exit(1);
}
hx_pid = start_thread(host_xmt, (void *) x);
if (hx_pid < 0) {
        perror("start_thread");
        exit(1);
}
c1_xpid = start_thread(c1_xmt, (void *) x);
if (c1_xpid < 0) {
        perror("start_thread");
        exit(1);
}
hr_pid = start_thread(host_rcv, (void *) x);
if (hr_pid < 0) {
        perror("start_thread");
        exit(1);
}
while(TRUE) { /* recieve from convert_1 sock */
        iRet = recvfrom(C_RCV_SOCK, BDL_REC[c].BUF, BUFFSIZE, 0,
                        (struct sockaddr*)&C_RCV_STR,&l_addr_size);
        BDL_REC[c].LEN = iRet;
        C_IN +=BDL_REC[c].BUF[iRet -1];
        while(BDL_LOCK) {
        }
        BDL_LOCK = TRUE;
        proc_bdl(c);
        BDL_LOCK = FALSE;
}
}

void host_xmt(void * data) {
  int addr_size = sizeof(struct sockaddr_in);
  u_int time_delay, l_len, iRet;
  u_char l_buf, l_cnt;
  u_short *read_ptr;
  char *write_ptr;
  struct timeval ta, tb;
  time_delay = 5000; /* start with a 5 mil_second delay...*/
  while(TRUE) {
        usleep(time_delay); /* wait between bundle transmissions */
        gettimeofday(&ta,0); /* get our reference start time */
        if (H_SND[CUR_SND_HBUF].CNT != 0) { /*do we need to send?*/
```

```
            l_buf = CUR_SND_HBUF;
            while(BDL_LOCK) {
            }
            CUR_SND_HBUF++; /* set up for bundle trans */
            if (CUR_SND_HBUF == MAX_OUT) CUR_SND_HBUF = 0;
            H_SND[CUR_SND_HBUF].LEN = 0; /* initialize for new */
            H_SND[CUR_SND_HBUF].CNT = 0; /* data bundles*/
            l_len = H_SND[l_buf].LEN;
            l_cnt = H_SND[l_buf].CNT;
            read_ptr = &H_SND[l_buf].END[0]; /* read 'endpt' buf */
            write_ptr = &H_SND[l_buf].BUF[l_len]; /* write data buf */
            memcpy(write_ptr,read_ptr,l_cnt*2);
            sprintf(H_SND[l_buf].BUF+l_len+l_cnt*2, /* write l_cnt */
                    "%c", l_cnt);
            iRet = sendto(H_SND_SOCK,H_SND[l_buf].BUF,
                          l_len+2*l_cnt+1,0,
                          (struct sockaddr*)&H_SND_STR,addr_size);
            H_MSG_CNT = 0; /* zero out congestion stats for next cycle */
            H_OUT +=l_cnt;
            DATA_VOL_H+=iRet;
            if (iRet == -1) {
                    printf("H SND sock send failed %d\n",errno);

            }
        }
        if (DATA_VOL_C1T > THRESH_C1T_UPPER) {
          if(C1_CONGEST < 6) C1_CONGEST++;
        }
        else if (DATA_VOL_C1T < THRESH_C1T_LOWER) {
          if(C1_CONGEST > 3) C1_CONGEST--;
        }
        DATA_VOL_C1T = 0; /* zero out stats for next cycle */
        gettimeofday(&tb,0);
        time_delay = 5000 - tb.tv_usec - ta.tv_usec;
        if (time_delay > 5000) time_delay = 5000; /* longest delay*/
        if (time_delay < 1) time_delay = 1; /* shortest delay */
  }
}
void c1_xmt(void * data) {
  int addr_size = sizeof(struct sockaddr_in);
  u_int time_delay, l_len, iRet, x;
  u_char l_buf, l_cnt, c1;
  u_short *read_ptr;
  char *write_ptr;
  struct timeval ta, tb;

  time_delay = 5000; /* start with a 5 mil_second delay...*/
  while(TRUE) {
        usleep(time_delay); /* wait between bundle transmissions */
        gettimeofday(&ta,0); /* get our reference start time */
        for (c1=1;c1<C1_CONV;c1++) {
          if (C_SND[CUR_SND_CBUF[c1]][c1].CNT != 0) { /*do we */
                l_buf = CUR_SND_CBUF[c1];     /*need to send msg?*/
                while(BDL_LOCK) {
                }
                CUR_SND_CBUF[c1]++; /* set up for bundle trans */
                if (CUR_SND_CBUF[c1] == MAX_OUT)
                  CUR_SND_CBUF[c1] = 0;
                C_SND[CUR_SND_CBUF[c1]][c1].LEN = 0; /* initialize */
                C_SND[CUR_SND_CBUF[c1]][c1].CNT = 0; /* data bundles */
                l_len = C_SND[l_buf][c1].LEN;
                l_cnt = C_SND[l_buf][c1].CNT;
                read_ptr = &C_SND[l_buf][c1].END[0];
                write_ptr = &C_SND[l_buf][c1].BUF[l_len];
                memcpy(write_ptr,read_ptr,2*l_cnt);
                sprintf(C_SND[l_buf][c1].BUF+l_len+l_cnt*2,
                        "%c", l_cnt);
                iRet = sendto(C_SND_SOCK[c1],C_SND[l_buf][c1].BUF,
                              l_len+2*l_cnt+1,0,
                              (struct sockaddr*)&C_SND_STR[c1],
                              addr_size);
                C_OUT+=l_cnt;
                C1_MSG_CNT = 0; /* zero out statistics */
                DATA_VOL_C1T+=iRet;
                if (iRet == -1) {
                  perror("C SND sock send failed");
                }
          }
        }
        if (DATA_VOL_H > THRESH_H_UPPER) {
          if(H_CONGEST < 6) {
                H_CONGEST++;
                while(BDL_LOCK) {
                }
                BDL_LOCK = TRUE;
                for(x=0;x<C1_CONV;x++) {
                  ctl_snd(BD_CAST,x,H_CONGEST); /* back off C1 */
                }
                memcpy(&BDL_REC[2].BUF[0], &H_CONGEST,1);
                proc_h_snd(BD_CAST, 0,22,2); /* back off Hosts */
                BDL_LOCK = FALSE;
            }
            /* based on what state we are in... */
            /* update host congetstion and send ctl */
          }
          else if (DATA_VOL_H < THRESH_H_LOWER) {
            if(H_CONGEST > 3) {
                    H_CONGEST--;
                    while(BDL_LOCK) {
                    }
                    BDL_LOCK = TRUE;
                    for(x=0;x<C1_CONV;x++) {
                      ctl_snd(BD_CAST,x,H_CONGEST); /* increase C1 */
                    }
                    memcpy(&BDL_REC[2].BUF[0], &H_CONGEST,1);
                    proc_h_snd(BD_CAST, 0,22,2); /*  increase Hosts */
                    BDL_LOCK = FALSE;
            }
          }
          DATA_VOL_H = 0;
          gettimeofday(&tb,0);
          time_delay = 5000 - tb.tv_usec - ta.tv_usec;
          if (time_delay > 5000) time_delay = 5000; /* longest delay*/
          if (time_delay < 1) time_delay = 1; /* shortest delay */
  }
}

void host_rcv(void * data) {
  int addr_size = sizeof(struct sockaddr_in);
  u_short iRet;
  u_char c = 1;
  while(TRUE) {
          iRet = recvfrom(H_RCV_SOCK, BDL_REC[c].BUF, BUFFSIZE, 0,
                                  (struct sockaddr*)&H_RCV_STR,&addr_size);
          H_IN+=BDL_REC[c].BUF[iRet - 1];
          BDL_REC[c].LEN = iRet;
          while(BDL_LOCK) {/* Pause while bdl process is locked */
          }
          BDL_LOCK = TRUE; /* lock it for our turn.. */
          proc_bdl(c);
          BDL_LOCK = FALSE; /* unlock for next proc */
  }
}

void update_mask(u_short h_mask, u_short l_addr, u_char l_act) {
  u_short A, l_len, tmp_addr;
  int x, tc, c1;
  if (l_act == 1) { /* add host mask to group addr */
          /* let's see if we need to request this IPmc address */
          if(IPMCDB[l_addr/GRP_TO_ADDR] == 0) {
            c1 = 1;
            tc = DATA_VOL_C1[c1];
            for(x = 1;x < C1_CONV;x++) {
                    if (DATA_VOL_C1[x] < tc) {
                      tc = DATA_VOL_C1[x];
                      c1 = x;
                    }
                    DATA_VOL_C1[x] = 0; /* zero out for next cycle */
            }
            ctl_snd(l_addr,c1,1);
            IPMCDB[l_addr/GRP_TO_ADDR] = c1;
          }
          A = MASKDB[l_addr];
          MASKDB[l_addr] = A | h_mask; /* A or h_mask*/
  }
  else { /* remove host mask from group addr */
          A = MASKDB[l_addr];
          MASKDB[l_addr] = A & ~h_mask; /* A and NOT h_mask */
          if (IPMCDB[l_addr/GRP_TO_ADDR] != 0) { /* a 'real' group */
            tmp_addr = l_addr;
            tmp_addr/=GRP_TO_ADDR; /* a cheap way to flush right */
            tmp_addr*=GRP_TO_ADDR; /* most bits from addr */
            for(x = 0;x<GRP_TO_ADDR;x++) {  /* now loop through the */
                    if(MASKDB[tmp_addr+x] != 0) /* database, seeif others*/
                      x = GRP_TO_ADDR + 10;   /* belong to the group*/
            }
            if (x < GRP_TO_ADDR + 1) { /* no other groups, so drop */
                    c1 = IPMCDB[l_addr/GRP_TO_ADDR];   /* from IPmc addr */
                    IPMCDB[l_addr/GRP_TO_ADDR] = 0;
                    ctl_snd(l_addr,c1,2);
            }
          }
  }
}

void ctl_snd(u_short l_addr, u_char c1, u_char l_act) {
  short l_len;    /*control msg routine*/
  u_char l_cnt;
  l_len = C_SND[CUR_SND_CBUF[c1]][c1].LEN;
  l_cnt = C_SND[CUR_SND_CBUF[c1]][c1].CNT;
```

```
    sprintf(C_SND[CUR_SND_CBUF[c1]][c1].BUF+l_len, "%c",l_addr/256);
    sprintf(C_SND[CUR_SND_CBUF[c1]][c1].BUF+l_len+1, "%c",l_addr);
    sprintf(C_SND[CUR_SND_CBUF[c1]][c1].BUF+l_len+2, "%c",l_act);
    C_SND[CUR_SND_CBUF[c1]][c1].LEN = l_len+22;
    C_SND[CUR_SND_CBUF[c1]][c1].CNT = l_cnt+1;
    C_SND[CUR_SND_CBUF[c1]][c1].END[l_cnt] = l_len+22;
}

void proc_c_snd(u_short l_addr, u_short l_ptr,
                u_short l_end, u_char b_p)
{ /* send c_1 msg */
    u_short l_len;
    u_char l_cnt, c1;
    char* w_ptr;
    char* r_ptr;
    c1 = IPMCDB[l_addr/GRP_TO_ADDR];
    if (c1 == 0) c1 = 1;
    l_len = C_SND[CUR_SND_CBUF[c1]][c1].LEN;
    l_cnt = C_SND[CUR_SND_CBUF[c1]][c1].CNT;
    sprintf(C_SND[CUR_SND_CBUF[c1]][c1].BUF+l_len, "%c",l_addr/256);
    sprintf(C_SND[CUR_SND_CBUF[c1]][c1].BUF+l_len+1, "%c",l_addr);
    w_ptr = &C_SND[CUR_SND_CBUF[c1]][c1].BUF[l_len+2];
    r_ptr = &BDL_REC[b_p].BUF[l_ptr];
    memcpy(w_ptr,r_ptr, l_end);
    C_SND[CUR_SND_CBUF[c1]][c1].LEN = l_len+l_end+2;
    C_SND[CUR_SND_CBUF[c1]][c1].CNT = l_cnt+1;
    C_SND[CUR_SND_CBUF[c1]][c1].END[l_cnt] = l_len+l_end+2;
    DATA_VOL_C1[c1]++;
}

void proc_h_snd(u_short gr_addr, u_short l_ptr,
                u_short l_end, u_char b_p)
{ /* send host msg */
    u_short l_len;
    u_char l_cnt;
    char* w_ptr;
    char* r_ptr;
    l_len = H_SND[CUR_SND_HBUF].LEN;
    l_cnt = H_SND[CUR_SND_HBUF].CNT;
    memcpy(&H_SND[CUR_SND_HBUF].BUF[l_len], &gr_addr,2);
    w_ptr = &H_SND[CUR_SND_HBUF].BUF[l_len+2];
    r_ptr = &BDL_REC[b_p].BUF[l_ptr];
    memcpy(w_ptr,r_ptr, l_end);
    H_SND[CUR_SND_HBUF].LEN = l_len+l_end+2;
    H_SND[CUR_SND_HBUF].CNT = l_cnt+1;
    H_SND[CUR_SND_HBUF].END[l_cnt] = l_len+l_end+2;
}

void proc_bdl(u_char b_p) {
    u_short x,l_len,start,len;
    u_char n;
    u_short r_mask, gr_addr, addr, end;
    r_mask = BDL_REC[b_p].BUF[0]*256+BDL_REC[b_p].BUF[1];
    start = 2;
    l_len = BDL_REC[b_p].LEN;
    n = BDL_REC[b_p].BUF[l_len-1];  /* n = # of messages in bundle */
    if(r_mask > 49150) { /* bundle came from WAN */
            DATA_VOL_C1[BDL_REC[b_p].BUF[1]]++;
            for (x=0;x < n;x++) {
                memcpy(&addr,&BDL_REC[b_p].BUF[start],2);
                memcpy(&end,&BDL_REC[b_p].BUF[l_len -1- 2*(n-x)],2);
                /*read end point of msg from 'end-section' of buffer */
                len = end - start - 2;
                gr_addr = MASKDB[addr];
                if (gr_addr != 0) { /* does anyone want this ?*/
                        if (((H_CONGEST<4)||(addr&1))||
                           ((!H_EXCEED)&&(H_CONGEST<6))) {
                           proc_h_snd(gr_addr,start+2,len, b_p);
                           if (H_CONGEST>3) {
                                   H_MSG_CNT++;
                                   if (H_MSG_CNT >
                                      H_CONGEST_ALLOWED - H_CONGEST)
                                      H_EXCEED = TRUE;
                           }
                        }
                }
                start = end;
            } /* loop through messages in bundle */
    } /* complete for WAN based traffic */
    else { /* must be local */
            for (x=0;x < n;x++) {
                memcpy(&addr,&BDL_REC[b_p].BUF[start],2);
                memcpy(&end,&BDL_REC[b_p].BUF[l_len -1- 2*(n-x)],2);
                /* offset from end for pointer */
                len = end - start - 2;
                if (len == 20) { /* control message */
                        update_mask(r_mask, addr, BDL_REC[b_p].BUF[start+2]);
                }
                else if(len > 1) {
                        proc_c_snd(addr,start+2,len, b_p);
```

```
                    gr_addr = MASKDB[addr] & ~r_mask; /* no reflection */
                    if (gr_addr != 0)
                       proc_h_snd(gr_addr,start+2,len, b_p);
                    start = end;
                }
            } /* loop through rest of messages */
    }
} /* end proc */


void init_prc_c2(void) {

   int iTmp, iRet, x, c;
   u_char cTmp;
   char c2_name[25];
   char *h_name = H_HOST;
   /* other c2 hosts are added here.. */
   struct hostent *hp, *gethostbyname();
   /* Start setting up for our recieve based socket...*/
   for (x=0;x<C1_CONV;x++) {
              C_SND[0][x].LEN = 0;
              C_SND[0][x].CNT = 0;
   }
   for(x=0;x<65535;x++) {
              MASKDB[x] = 0;
              IPMCDB[x] = 0;
   }
   H_RCV_SOCK = socket(AF_INET,SOCK_DGRAM,0);
   if (H_RCV_SOCK == INVALID_SOCKET) {
              printf ("H_RCV_socket() failed, Err: %d\n", -1);
   }
   /* name the socket */
   H_RCV_STR.sin_family      = AF_INET;
   H_RCV_STR.sin_addr.s_addr = htonl(INADDR_ANY);
   H_RCV_STR.sin_port        = htons(H_RCV_PORT);
   iRet = bind(H_RCV_SOCK, (struct sockaddr*) &H_RCV_STR,
                       sizeof(H_RCV_STR));
   if (iRet == SOCKET_ERROR) {
              printf ("H_RCV_bind() failed, Err: %d\n",-1);
   }
   /* so far so good, now lets set up our C1 based socket...*/

   C_RCV_SOCK = socket(AF_INET,SOCK_DGRAM,0);
   if (C_RCV_SOCK == INVALID_SOCKET) {
              printf ("C_RCV_socket() failed, Err: %d\n", -1);
   }
   /* name the socket */
   C_RCV_STR.sin_family      = AF_INET;
   C_RCV_STR.sin_addr.s_addr = htonl(INADDR_ANY);
   C_RCV_STR.sin_port        = htons(C_RCV_PORT);
   iRet = bind(C_RCV_SOCK, (struct sockaddr*) &C_RCV_STR,
                       sizeof(C_RCV_STR));
   if (iRet == SOCKET_ERROR) {
              printf ("H_RCV_bind() failed, Err: %d\n",-1);
   }

   /* now set up the C1 send socket */
   for (x=1;x<C1_CONV;x++) {
              switch(x) {
              case 1: sprintf(c2_name,"%s", C1_HOST_A);
                 break;
              case 2: sprintf(c2_name,"%s", C1_HOST_B);
                 break;
              default:
                 printf("wrong C1_CONV name!\n");
              }
              C_SND_SOCK[x] = socket(AF_INET,SOCK_DGRAM,0);
              if (C_SND_SOCK[x] == INVALID_SOCKET) {
                 printf ("CSND_socket() failed, Err: %d\n", -1);
                 exit(1);
              }
              /* name the socket */
              hp = gethostbyname(c2_name);
              if (hp == 0) {
                 printf(" %s unknown host\n",c2_name);
                 exit(2);
              }
              memcpy( (char *)&C_SND_STR[x].sin_addr, (char*)hp->h_addr,
                        hp->h_length);
              C_SND_STR[x].sin_family =    AF_INET;
              C_SND_STR[x].sin_port =      htons(C_SND_PORT);
   }
   /* set up host send socket */
     H_SND_SOCK = socket(AF_INET,SOCK_DGRAM,0);
   if (H_SND_SOCK == INVALID_SOCKET) {
              printf ("HSND_socket() failed, Err: %d\n", -1);
              exit(1);
   }
   /* name the socket */
```

```
  hp = gethostbyname(h_name);
  if (hp == 0) {
            printf(" unknown host\n");
            exit(2);
  }
  memcpy( (char *)&H_SND_STR.sin_addr, (char*)hp->h_addr,
              hp->h_length);
  H_SND_STR.sin_family =      AF_INET;
  H_SND_STR.sin_port =        htons(H_SND_PORT);
}

void status(void * data) {
  while(TRUE) {
            sleep(1);
            printf("H_IN %d  H_OUT %d  C_IN %d  C_OUT %d \n",
                 H_IN, H_OUT, C_IN, C_OUT);
            H_IN = 0;
            H_OUT = 0;
            C_IN = 0;
            C_OUT = 0;
  }
}
```

Host Process

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/prctl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>

#define BUFFSIZE 65536 /* maximum buffer size */
#define MAX_OUT 30 /* maximum outstanding bundle transmissions is 30 */
#define MAX_MSG 150 /* maximum number of messages per bundle...*/
#define SND_ADDR "234.4.1.1"
#define RCV_ADDR "234.3.1.1"
#define CONGEST_ALLOWED 7 /* allow x-CONGEST messages if moderate congestion */
#define MASK 4 /* host's mask for message routing (2^k) ...*/
#define GRP_TO_ADDR 32 /* group to IPmc address conversion */
#define TTL_VALUE 1
#define RCV_PORT 3459
#define SND_PORT 3458
#define TRUE 1
#define FALSE 0
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
#define STRING "1234567890aaaaaaaaaa1234567890bbbbbbbbbb1234567890cccccccccc"

struct b_buffer { /* built as struct for array ref... */
   char *BUF;
   u_short LEN; /* length of current buffer in bytes */
   u_char CNT; /* number of messages within buffer */
   u_short *END; /*record where each msg ends*/
   u_short LOCK;
}BDL_REC, SND[256];

char *MSG_BUF;
char *UPDATE;
/* allow for outstanding bundle transmissions */

struct sockaddr_in RCV_STR, SND_STR; /*socket stuctures */
struct ip_mreq stMreq;

/* set up globals (ALL CAPS) for sproc processes */
u_char CUR_RCV_BBUF, CUR_SND_BBUF;
u_short CONGEST, SEND_CNT, RCV_CNT, RSEND_CNT, MSG_CNT;
int RCV_SOCK, SND_SOCK;
int DEL_TIME, DELAY, AVG_DEL, MAX_DEL, DEL_I;
char CTMP, HOLD_FLAG, EXCEED;
char *MSG_BUFFPTR;
FILE *HOST_FILE;


void xmt(void);

void ctl_snd(u_short l_addr, u_char l_act);

void snd(u_short l_addr, u_short l_end);

void proc_bdl(void);

void init_prc_host(int argc, char *argv[]);

void test_proc(void);

void demo(void);

void timep(void);

void main(int argc, char* argv[]) {

   int xpid;
   int addr_size = sizeof(struct sockaddr_in);
   u_short iRet;
   BDL_REC.BUF = (char *) malloc(BUFFSIZE);
   BDL_REC.END = (u_short *) malloc(2*MAX_MSG);
   MSG_BUF = (char *) malloc(BUFFSIZE);
   UPDATE = (char *) malloc(BUFFSIZE);
   for(xpid=0;xpid<256;xpid++) {
```

```
			SND[xpid].BUF = (char *) malloc(BUFFSIZE);
			SND[xpid].END = (u_short *) malloc(2*MAX_MSG);
	}
	init_prc_host(argc, argv);
	xpid = sproc((void (*)(void *))xmt, PR_SALL, 0);
	xpid = sproc((void (*)(void *))test_proc, PR_SALL, 0);
	xpid = sproc((void (*)(void *))demo, PR_SALL, 0);
	xpid = sproc((void (*)(void *))timep, PR_SALL, 0);
	while(TRUE) {
			iRet = recvfrom(RCV_SOCK, BDL_REC.BUF, BUFFSIZE, 0,
				(struct sockaddr*)&RCV_STR,&addr_size);
			BDL_REC.LEN = iRet;
			proc_bdl();
	}
}

void xmt(void) {
	int addr_size = sizeof(struct sockaddr_in);
	u_short time_delay, l_len;
	int iRet;
	struct timeval ta, tb;
	u_char l_buf, l_cnt;
	time_delay = 5000; /* start with a 5 mil_second delay...*/
	while(TRUE) {
			usleep(time_delay); /* wait between bundle transmissions */
			gettimeofday(&ta);  /* get our reference start time */
			if (SND[CUR_SND_BBUF].CNT != 0) { /*do we need to send?*/
			  l_buf = CUR_SND_BBUF;
			  while(HOLD_FLAG) { /* wait until released */
			  }
			  CUR_SND_BBUF++; /* set up for bundle trans */
			  if (CUR_SND_BBUF > MAX_OUT - 1) CUR_SND_BBUF = 0;
			  SND[CUR_SND_BBUF].LEN = 2; /* initialize for new */
			  SND[CUR_SND_BBUF].CNT = 0; /* data bundles */
			  l_len = SND[l_buf].LEN;
			  l_cnt = SND[l_buf].CNT;
			  /* copy end points into buffer */
			  while(SND[l_buf].LOCK) {
			  }
			  SND[l_buf].LOCK = TRUE;
			  memcpy(&SND[l_buf].BUF[l_len],
					&SND[l_buf].END[0], 2*l_cnt);
			  /* copy the message count as last character */
			  memcpy(&SND[l_buf].BUF[l_len+l_cnt*2],
					&l_cnt,1);
			  SND[l_buf].LOCK = FALSE;
			  iRet = sendto(SND_SOCK,SND[l_buf].BUF,
						l_len+2*l_cnt+1,0,
					(struct sockaddr*)&SND_STR,addr_size);
			  SEND_CNT+=l_cnt;
			  MSG_CNT=0;
			  EXCEED = FALSE; /* reset the EXCEED FLAG */
			  if (iRet < 0) {
					printf("SND sock send failed %d\n",errno);
			  }
			}
			gettimeofday(&tb);
			time_delay = 5000 - tb.tv_usec - ta.tv_usec;
			if (time_delay > 5000) time_delay = 5000; /* longest delay*/
			if (time_delay < 1) time_delay = 1; /* shortest delay */
	}
}

void ctl_snd(u_short l_addr, u_char l_act) { /* send control msg */
	u_short l_len;
	u_char l_cnt;
	l_len = SND[CUR_SND_BBUF].LEN;
	l_cnt = SND[CUR_SND_BBUF].CNT;
	while(SND[CUR_SND_BBUF].LOCK) {
	}
	SND[CUR_SND_BBUF].LOCK = TRUE;
	memcpy(&SND[CUR_SND_BBUF].BUF[l_len],&l_addr,2);
	memcpy(&SND[CUR_SND_BBUF].BUF[l_len+2],&l_act,1);
	SND[CUR_SND_BBUF].LEN = l_len+22;
	SND[CUR_SND_BBUF].CNT = l_cnt+1;
	SND[CUR_SND_BBUF].END[l_cnt] = l_len+22;
	SND[CUR_SND_BBUF].LOCK = FALSE;
}

void snd(u_short l_addr, u_short l_end) { /* send msg */
	u_short l_len;
	u_char l_cnt;
	char *read_p;
	char *write_p;
	u_short *short_p;
	/* this next logic statement says,
	 * IF [NO CONGESTION or PRIORITY MESSAGE] OR [WE HAVEN'T EXCEEDED
	 * OUR LIMIT and WE'RE BELOW MAX CONGESTION]
	 * this will allow for a message to be sent in the following cases
	 * 1. No Congestion (ie CONGEST = 3)
 * 2. This is a priority message, ALWAYS TRANSMIT THESE!!
 * 3. Congestion is below max and we haven't exceeded our quota of msgs
 */
 if (((CONGEST<4)||(l_addr&1))||((!EXCEED)&&(CONGEST<6))) {
        HOLD_FLAG = TRUE;
        l_len = SND[CUR_SND_BBUF].LEN;
        l_cnt = SND[CUR_SND_BBUF].CNT;
        short_p = &l_addr;
        write_p = &SND[CUR_SND_BBUF].BUF[l_len];
        while(SND[CUR_SND_BBUF].LOCK) {
        }
        SND[CUR_SND_BBUF].LOCK = TRUE;
        memcpy(write_p,short_p, 2); /* write the msg address */
        write_p +=2; /* start writing after the address */
        read_p = &MSG_BUF[0];
        memcpy(write_p, read_p, l_end);
        SND[CUR_SND_BBUF].LEN = l_len+l_end+2;
        SND[CUR_SND_BBUF].CNT = l_cnt+1;
        SND[CUR_SND_BBUF].END[l_cnt] = l_len+l_end+2;
        SND[CUR_SND_BBUF].LOCK = FALSE;
        HOLD_FLAG = FALSE;
        if (CONGEST>3) {
          MSG_CNT++;
          if (MSG_CNT > CONGEST_ALLOWED - CONGEST)
                EXCEED = TRUE;
        }
 }
}

void proc_bdl(void) {
  u_short addr, l_len, start, end, len;
  u_short x, i;
  u_char l_cnt;
  struct timeval t, tr;
  start = 0;
  l_len = BDL_REC.LEN;
  l_cnt = BDL_REC.BUF[l_len-1];
  RCV_CNT +=l_cnt;
  for (x=0;x<l_cnt;x++) {
        memcpy(&addr, &BDL_REC.BUF[start],2);
        memcpy(&end, &BDL_REC.BUF[l_len - 2*(l_cnt-x) -1],2);
        if (addr & MASK) { /* msg for us? */
          len = end - start - 2;
          if (len==22) {
                CONGEST = BDL_REC.BUF[start+2];
          }
          if(len == 138) {
                gettimeofday(&t);
                memcpy(&tr, &BDL_REC.BUF[start+3], sizeof(struct timeval));
                DELAY = -1*((tr.tv_sec - t.tv_sec)*1000000 +
                  tr.tv_usec - t.tv_usec + DEL_TIME);
                if (DELAY > 0) {
                  DEL_I++;
                  if (DELAY > MAX_DEL) MAX_DEL = DELAY;
                  AVG_DEL+=DELAY;
                }
          }
          /* a potential command may look like
            memcpy(UPDATE,BDL_REC.BUF+start+2,len);
            to get the message to update the data base... */
        }
        start = end;
  } /* loop through messages in bundle */
} /* complete traffic */


void init_prc_host(int argc, char *argv[]) {

  int iTmp;
  u_short mask;
  int iRet;
  u_char x;
  struct hostent *hp;
  mask = MASK;
  HOLD_FLAG = FALSE;
  BDL_REC.LOCK = FALSE;
  for(iTmp = 0;iTmp < 256;iTmp++) {
        SND[iTmp].LOCK = FALSE;
  }
  CUR_SND_BBUF = 0;
  CONGEST = 3;
  for (x = 0; x < MAX_OUT;x++) {
        SND[x].LEN = 2; /* initialize for new */
        SND[x].CNT = 0; /* data bundles */
        memcpy(SND[x].BUF, &mask, 2);
  }
  /* Start setting up for our recieve based socket...*/
  RCV_SOCK = socket(AF_INET,SOCK_DGRAM,0);
  if (RCV_SOCK == INVALID_SOCKET) {
        printf ("RCV_socket() failed, Err: %d\n", -1);
```

```
            exit(1);
    }
    /* name the socket */
    RCV_STR.sin_family      = AF_INET;
    RCV_STR.sin_addr.s_addr = htonl(INADDR_ANY);
    RCV_STR.sin_port        = htons(RCV_PORT);
    iRet = bind(RCV_SOCK, (struct sockaddr*) &RCV_STR,
                    sizeof(RCV_STR));
    if (iRet == SOCKET_ERROR) {
            printf ("RCV_bind() failed, Err: %d\n",-1);
    }

    /* so far so good, now lets set up our send socket...*/
    SND_SOCK = socket(AF_INET,SOCK_DGRAM,0);
    if (SND_SOCK == INVALID_SOCKET) {
            printf ("SND_socket() failed, Err: %d\n", -1);
            exit(1);
    }
    /* name the socket */

    hp = gethostbyname(argv[1]);
    if (hp == 0) {
            printf(" unknown host\n");
            exit(2);
    }
    memcpy( (char *)&SND_STR.sin_addr, (char*)hp->h_addr,
              hp->h_length);
    SND_STR.sin_family =        AF_INET;
    SND_STR.sin_port =          htons(SND_PORT);
}

/* this is strictly for testing the code... */
void test_proc(void) {
    struct db {
            int repeat;
            int addr;
            int l_len;
    } demo[1000];
    int r, x, c, repeat, l_len,addr, end;
    u_char ctmp;
    sprintf(MSG_BUF, "%s",STRING);
    HOST_FILE = fopen("host.fil" , "rb");
    if(HOST_FILE==NULL)
            printf("Host file error!\n");
    c = fscanf(HOST_FILE, "%u", &end);
    for (x = 0;x < end;x++) {
            c = fscanf(HOST_FILE, "%u %u %u \n",
                            &repeat, &addr, &l_len);
            demo[x].repeat = repeat;
            demo[x].addr = addr;
            demo[x].l_len = l_len;
    }
    fclose(HOST_FILE);
    while (TRUE) {
            for (x = 0;x < end;x++) {
              sginap(2);
              for (r = 0;r < demo[x].repeat; r++) {
                      ctmp = demo[x].l_len;
                      if (demo[x].l_len == 1) {
                        ctmp = 1;
                        ctl_snd(demo[x].addr, ctmp);
                      }
                      else if (demo[x].l_len == 2) {
                        ctmp = 2;
                        ctl_snd(demo[x].addr, ctmp);
                      }
                      else {
                        snd(demo[x].addr, demo[x].l_len);
                      }
              }
            }
    }
}

void demo(void) {
  int AVG_DIS;
  DEL_I = 1;
  while(TRUE) {
            sginap(50);
            AVG_DIS = AVG_DEL / DEL_I;
            printf("Sent %d,  Recv %d   Msgs / sec Delay (Avg Max) (ms) %d %d\n",
                 SEND_CNT*2, RCV_CNT * 2, AVG_DIS/1000, MAX_DEL/1000);
            SEND_CNT = 0;
            RCV_CNT = 0;
            RSEND_CNT = 0;
            MAX_DEL = 0;
            AVG_DEL = 0;
            DEL_I = 1;
  }
}
```

```
void timep(void) {
  int t_sock, iRet;
  int addr_size = sizeof(struct sockaddr_in);
  struct timeval t, t1;
  struct sockaddr_in t_str;
    /* Start setting up for our recieve based socket...*/
  t_sock = socket(AF_INET,SOCK_DGRAM,0);
  if (t_sock == INVALID_SOCKET) {
          perror("t_socket() failed,");
          exit(1);
  }
  /* name the socket */
  t_str.sin_family      = AF_INET;
  t_str.sin_addr.s_addr = htonl(INADDR_ANY);
  t_str.sin_port        = htons(3429);
  iRet = bind(t_sock, (struct sockaddr*) &t_str,
                     sizeof(t_str));
  if (iRet == SOCKET_ERROR) {
          perror("time_bind() failed");
  }
  while(TRUE) {
          iRet = recvfrom(t_sock, &t, sizeof(struct timeval), 0,
              (struct sockaddr*)&t_str,&addr_size);
          gettimeofday(&t1);
          DEL_TIME = (t1.tv_sec - t.tv_sec)*1000000 + t1.tv_usec - t.tv_usec;
  }
}
```

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE MESSAGE-BUNDLE CONVERTING IN IP MULTICAST-BASED HIGH LEVEL ARCHITECTURE EXERCISES	5. FUNDING NUMBERS
6. AUTHOR(S) Tracy A. Bobo, Captain USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Department of Defense is pushing for more wide-spread and realistic interactive training simulations which increases the demand on network capacity and resources. While network bandwidth is a measurable resource, packet bandwidth, or the number of packets-per-second (Pk/s) a host can handle, is a shifting commodity. This research analyzes host performance characteristics under varying data loads. The hosts include SGI single and multi-processor systems and Intel Pentium platforms using both Windows 95 and Linux Operating Systems. The networking media covers Ethernet, ATM and FDDI. For the ATM network, both AAL5 and IP over ATM were analyzed. With the data from this research, a system is proposed and developed that takes individual messages and bundles them into multi-message packets. This bundling process overcomes the 5,000 Pk/s limitation, reduces the CPU network handling time and introduces a flow-control mechanism at the local network level.

While the idea of bundling messages to increase CPU efficiency is not new, there are no current methods of bundling within the new High Level Architecture (HLA). This proposed process is a novel approach to introduce flow control, priority message handling and increase address space while utilizing bundled data delivery. For traditional network delivery, typical CPU usage from network data varies as a function of traffic load, ranging from 5% at 500 messages-per- second to over 80% at 4,000 messages-per-second. The new bundling process requires 10% at 500 messages-per-second but only increases to 13% at 4,000 messages-per-second.

14. SUBJECT TERMS			15. NUMBER OF PAGES 122
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL