

University of Central Florida

**STARS**

---

Institute for Simulation and Training

Digital Collections

---

1-1-1996

## Application Level Hardware Filtering For DIS: Final Report

Udaya B. Vemulapati

Find similar works at: <https://stars.library.ucf.edu/istlibrary>

University of Central Florida Libraries <http://library.ucf.edu>

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### Recommended Citation

Vemulapati, Udaya B., "Application Level Hardware Filtering For DIS: Final Report" (1996). *Institute for Simulation and Training*. 18.

<https://stars.library.ucf.edu/istlibrary/18>

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-94-C-0024  
CDRL A00X  
STRICOM  
February 27, 1996

# Application Level Hardware Filtering For DIS

## Final Report

Institute for Simulation and Training  
3280 Progress Drive  
Orlando FL 32826

University of Central Florida  
Division of Sponsored Research



IST-TR-96-16

B315

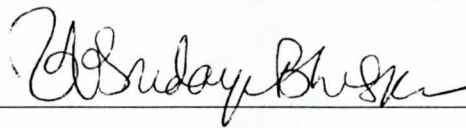
# Application Level Hardware Filtering For DIS

## Final Report

IST-TR-96-16  
February 27, 1996

Prepared For:  
STRICOM  
Contract Number N61339-94-C-0024  
CDRL A00X

Author:  
Dr. Udaya B Vemulapati



Reviewed By:  
Scott H. Smith



## Table of Contents

Abstract .....	2
1.0 Purpose .....	3
1.1 Introduction .....	3
2.0 Bandwidth Reduction Techniques .....	3
3.0 DIS Protocols .....	4
4.0 Packet Flow -- The Inside View .....	5
5.0 Hardware Filter .....	5
6.0 Application Level Interface (API) .....	7
7.0 Possible Hardware Architecture .....	8
7.1 Microcode Bit Fields, Instructions, and Timing Estimates .....	9
8.0 A Typical DIS Exercise .....	12
9.0 Experimental Results .....	12
9.1 Rectangular Paths .....	12
9.2 Random Path .....	14
9.3 Actual Simulation Data Replayed .....	15
9.4 Radion Transmissions .....	18
10.0 Packet Processing Overhead .....	19
11.0 Run Time Overhead .....	20
12.0 Conclusions .....	21
13.0 Acknowledgments .....	21
14.0 References .....	22
Appendix A: API Function Prototypes for Stages 1, 2, 3 .....	23
Appendix B: API Function Prototypes for the Application Stage of the Filter .....	24
Appendix C: Main Loop Code of ISTCGF .....	26
Appendix D: API Function Calls Code .....	26



## ABSTRACT

As DIS exercises grow in size and complexity, the volume of data each entity receives increases. At the same time, less of this information is typically useful to any particular entity (especially activity outside the entities' vision of interest). Currently, each application running on any processor connected to an Ethernet LAN will be receiving all the PDU elements and acts or discards after examining the contents. In this paper we describe our effort to study the feasibility of filtering these PDU elements right at the Ethernet entry point based on the requirements specified by the application. The filtering is done by a *smart card* that is placed between the Ethernet card and the system bus (or more conveniently by replacing the entire Ethernet card with this new card). We describe an Application Programming Interface (API) specification that the application uses to specify the filtering requirements (as to what types of PDU packets can be filtered). We also provide a high-level description of the filter as well as a design. We present the results of simulation experiments that analyze the effectiveness and usefulness of this filter. Based on the results of the experiments, we conclude that this technique would enable simulators of limited capacity to participate in large-scale DIS exercises by reducing the amount of data that the applications actually see.

## 1.0 Purpose

This report is a deliverable item (CDRL A00X) required in completion of subtask 3.4.2.3, "APPLICATION LEVEL HARDWARE FILTERS", on STRICOM contract N61339-94-C-0024 entitled, "TRIDIS: A Testbed for Research in Distributed Interactive Simulation.

### 1.1 Introduction

Distributed Interactive Simulation (DIS) is a set of protocols to permit the linking of various types of simulations at multiple locations (possibly spread over a vast geographical region) to create a complex, realistic simulation environment. The individual simulations can be one of many types, such as: strictly computer controlled virtual entities (computer generated forces), live operator controlled virtual entities (human in-the-loop simulators), and live entities (actual operational platforms and evaluation systems).

In operation, each of the entities participating in the simulation will periodically issue information packets, called Protocol Data Units (PDUs), as its operating state changes. The other entities will monitor the network and retrieve these packets and use the information to update their internal model of each of the distant entities (distant in the real world, though not necessarily in the simulated world).

Most of the DIS exercises that have been run to date, have used the broadcast mode of the Ethernet for transmission of the PDUs. This sends all information to all entities and each entity is required to examine each PDU and determine if the data it contains is relevant to its mission. As more entities participate in an exercise and transmit their state data, a lesser percentage of the information received by each entity will be of use to it. This increasing amount of irrelevant data will reduce the performance of both the network and the individual applications that have to process it. It is very desirable to reduce the irrelevant data seen by the network and applications.

## 2.0 Bandwidth Reduction Techniques

Many different approaches are being studied to control the increased data flow as DIS exercises grow from dozens, to hundreds, to thousands of entities. The ultimate goal is to be able to have tens of thousands of entities participating in the same simulation exercise.

Bassiouni, Chiu, and Williams [2] discuss algorithms for reducing network traffic by filtering performed by network gateways. Each gateway maintains accurate information about entities in its local LAN as well as entities on other LANs. Each gateway will only send relevant data to foreign LANS that require it, called filtering at transmission, and only propagate information to local nodes that require it, called filtering at reception.

Kerr, and Dobosz [3], suggest that putting different PDU families on different UDP ports will reduce the time applications spend filtering as they can ignore groups they are not interested in.

Russo, Schuette, Smith, and McGuire [4], Pullen, and White [5], Van Hook, Rak, and Calvin [6] and others, are analyzing the use of multicast groups to reduce both network traffic and individual applications irrelevant data loads. The current concept is to divide the virtual world into grids and entities would subscribe to the multicast groups that represent the grid squares that they are interested in. Current estimates are that thousands or tens of thousands of multicast groups may be required to effectively partition the virtual world. There are still more questions on how this might be implemented than there are answers.

At the Institute for Simulation and Training, some current applications running on IBM PC compatible hardware were found to be discarding (because of queue overflow) 70-80% of the PDU packets received, because there was insufficient computational power to process them all.

### 3.0 DIS Protocols

The IEEE standard 1278.1 [1] defines 27 PDU types in six PDU families. Several new types have been proposed since. As the protocol matures and other non-military applications are devised, additional families and types will be developed. It is important that every new application that is written, does not develop its own unique PDU types and protocols. This would severely limit the interoperability of various applications and make constructing large virtual environments nearly impossible. Figure 1 shows the currently defined PDUs by family and type.

- a) Entity Information/ Interaction
  - 1) Entity State PDU
  - 2) Collision PDU
- b) Warfare
  - 1) Fire PDU
  - 2) Detonation PDU
- c) Logistics
  - 1) Service Request PDU
  - 2) Resupply Offer PDU
  - 3) Resupply Received PDU
  - 4) Resupply Cancel PDU
  - 5) Repair Complete PDU
  - 6) Repair Response PDU
- d) Simulation Management
  - 1) Start/Resume PDU
  - 2) Stop/Freeze PDU
  - 3) Acknowledge PDU
  - 4) Action Request PDU
  - 5) Action Response PDU
  - 6) Data Query PDU
  - 7) Set Data PDU
  - 8) Data PDU
  - 9) Event Report PDU
  - 10) CommentMessage PDU
  - 11) Create Entity PDU
  - 12) Remove Entity PDU
- e) Distributed Emission Regeneration
  - 1) Electromagnetic Emission PDU
  - 2) Designator PDU
- f) Radio Communications
  - 1) Transmitter PDU
  - 2) Signal PDU
  - 3) Receiver PDU

Figure 1: Currently Defined PDUs by Family and Type



## 4.0 Packet Flow — The Inside View

The International Standards Organization (ISO) has developed a seven layer model for computer systems to communicate with each other: Physical layer, Data-Link layer, Network layer, Transport layer, Session layer, Presentation layer, and Application layer. In the ISO model, each layer, except the Physical layer, adds its own header information and the data-link layer also adds trailer information. The Ethernet hardware board encapsulates the lower (first) two layers. The Internet Protocol (IP) layer is the network layer and the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are two different Transport layers. TCP uses IP to transport a reliable stream of information between two processes [3]. The UDP is an unreliable, connection-less transport protocol [8]. TCP packets are guaranteed to arrive in order and are retransmitted if they don't. UDP packets are sent using the best-effort of the network, but are not guaranteed to arrive and are not retransmitted if they are dropped. DIS applications use the UDP for most PDU transmissions because the PDUs are resent periodically anyway and an occasional dropped packet will be updated when the next one is transmitted. When a DIS application sends a PDU message over the Internet, the transport layer (UDP) adds 8 header bytes, the network layer (IP) adds 20 header bytes, and the data-link layer (Ethernet) adds 14 header and 4 trailer bytes to the original PDU. When this PDU is received by the destination computer, this process is reversed and the Ethernet layer removes its 14 header and 4 trailer bytes, the IP layer removes its 20 header bytes and the UDP layer removes its 8 header bytes. When a PDU packet arrives that is not needed by the application, it must still be processed through all of these layers and finally be discarded by the application.

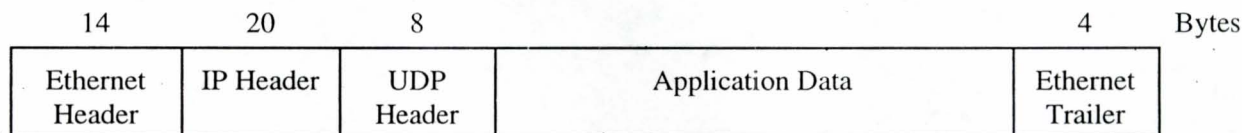


Figure 2: Packet Contents on the Ethernet

## 5.0 Hardware Filter

This paper examines the advantages of designing a hardware device to off load much of the filtering tasks from the CPU. This device might be implemented as a redesigned Ethernet card to replace the existing card, or as a device to be inserted at the transceiver cable interface, between the controller electronics, and the transceiver. This device will only address the problem of reducing the irrelevant data reaching an individual application, not the problem of reducing the network traffic.

In the simulations performed, the filter tasks were divided into different layers that correspond to the different layers of the Internet protocol stack. Ultimately, each packet examined by the hardware filter is either sent on to the DIS application or discarded. The following discussion uses three states to identify the disposition of the packet. The states are:

- Filter** the final disposition is not yet known, the packet is sent to the next stage of the filter.
- Bypass** this packet is to be kept, bypass the remaining filter stages and send it to the application.
- Discard** this packet is not useful to the application, throw it away.

Figure 4 shows the overall flow of packets through the filter.

In keeping with the DIS philosophy of discarding old information, the filter parameters each have a time tag. If the DIS application does not update the filter parameters before the timeout, (12-14 seconds) the filter ignores those parameters and *bypasses* those packets.

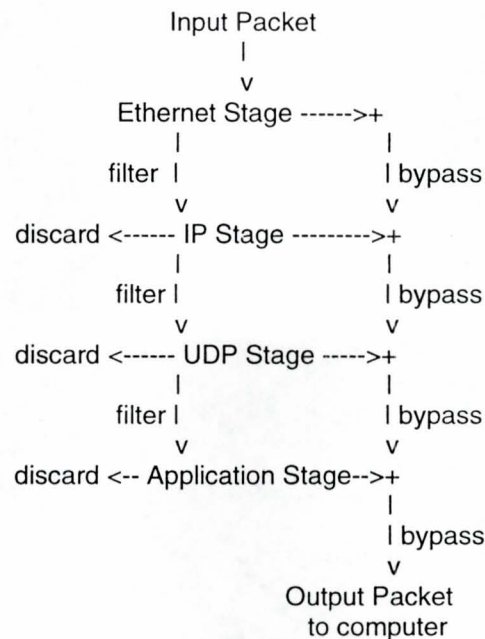


Figure 4: Overall flow of packets through the filter

The Ethernet layer identifies the Ethernet address as belonging to one of three classes: broadcast, multicast, or other. Each of the three address classes can be individually *filtered* or *bypassed*. (Note: the filter doesn't actually check for a match with the Ethernet port address of this node. This would be redundant since the Ethernet hardware does this comparison).

The IP layer *bypasses* all non-UDP packets and packets whose IP address is not broadcast or multicast. Packets with broadcast or multicast addresses are optionally *discarded*.

The UDP layer checks for a match on a list of Destination Port Numbers that identify PDU ports. PDU Packets are then either *filtered* or *bypassed* and the non-PDU packets are either *bypassed* or *discarded*.

The application layer is the most complicated and checks for matches with various parameters in the different fields of the different PDU types, including ignoring some PDU types entirely. The API section covers the details on the filter functions for the various PDU types.

This filter doesn't actually implement all of the functionality of the different protocol layers. This functionality will still have to be performed by the system software for those packets that are passed. Time will be saved by not processing the undesired packets through all of the protocol layers to merely discard them at the application level.



## 6.0 Application Level Interface (API)

The application communicates with the filter hardware through a driver program. Only the driver program communicates directly with the hardware. Appendix A is the header file for the API function prototypes for the first three stages of the filter. The API function prototypes for the application stage are in Appendix B.

The PDU Header Stage checks for a match to the DIS protocol version and exercise id and discards packets that don't match.

Each PDU type could potentially have a different type of filtering function to determine whether the PDU is useful to the simulation application. We chose to keep these numbers to a minimum (fewer cases than PDU types). All of the PDUs in the Logistics family are grouped together, as are the PDUs in the Simulation Management family. These PDU types don't produce a significant percentage of the PDU traffic, so don't warrant complex treatment. The PDUs in both of these groups are typically addressed to a specific entity site and application or to all sites and applications. The API for these two families allows for passing only those PDU addressed to the running application, and optionally to ignore PDUs addressed to all sites and applications. Fire and designator PDUs use nearly the same filter function as the logistics and simulation management families. The only difference is to allow for ignoring PDUs addressed to unknown sites and applications, rather than all sites and applications. (it doesn't make sense for fire or designator PDUs to target all sites and applications)

Entity State PDUs account for a significant percentage of the total PDU traffic and therefore warrant fairly significant filtering techniques. The filter allows seven different ranges for different classes of entities. The entities are classified by their approximate speed. (see Appendix B, enumeration type `KIND_DOMAIN_SPEED`). Additionally, the filter can maintain different ranges for each of 30 own ship vehicles. To eliminate the need for floating point hardware, these admittance regions are treated as cubes rather than spheres. This allows simple fixed point comparisons on each of the x, y, and z components. (the IEEE floating point standard was specifically designed to allow floating point comparisons to be performed with the same fixed point comparison hardware) Additionally, only 32 bit floating point ranges are used. This also simplifies the hardware. 32 bit IEEE floating point numbers allow a precision of approximately 1/4 of a meter for numbers of the magnitude of the radius of the earth. This is plenty accurate for this course range test. The 64 bit entity positions are easily converted to 32 bit numbers by simply ignoring some of the bits. (easy in hardware).

Transmitter PDUs are filtered based on the transmitter frequency and bandwidth being within one of a set of frequency and bandwidth limits.

Signal PDUs have only entity id and radio id available to filter on. The filter-able information (frequency, bandwidth) is in the corresponding transmitter PDU. The application sends the filter a list of entity and radio ids for which to pass Signal PDUs.

Emission and receiver PDUs are either bypassed or discarded.

Collision and detonation PDUs generally occurs at such low rates, it was decided to just bypass them. If in some particular exercise, they become significant, they could be handled similarly to fire and designator PDUs.

## 7.0 Possible Hardware Architecture

The filter algorithms to be implemented require quite a lot of field comparisons performed at the different levels of the protocol stack. In many cases, multiple comparisons need to be made against the same field to pass or reject multiple types. Fixing the particular fields and the number of values to be tested in each field would make the hardware quite inflexible. A programmable architecture would be able to accommodate new or modified PDU types as the DIS specification matures. A programmable architecture would also allow individual applications to tune the filter to a greater degree, thereby eliminating a greater percentage of the irrelevant information. The application might be able to reduce or even eliminate its internal filtering on some PDU types, further improving application efficiency.

A microcode programmable sequencer can meet all of these goals.

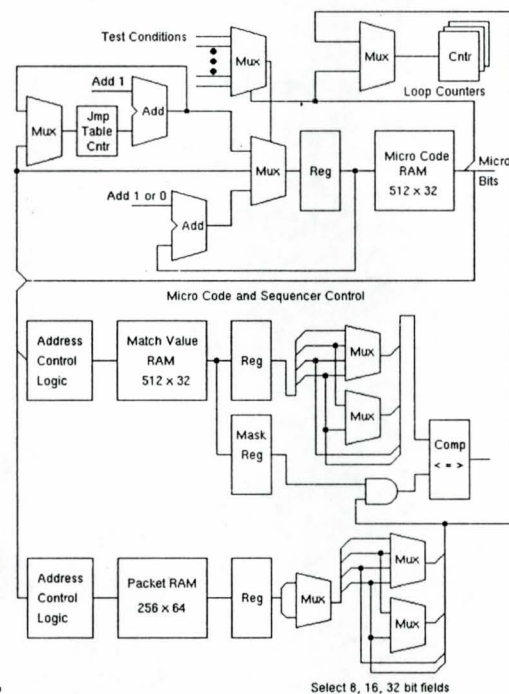


Figure 5: Block Diagram of Filter Hardware

As the match values must be updated frequently, they will be stored in a separate memory from the actual microcode for the sequencer. The sequencer will have a three way loop/jump/continue instruction to facilitate matches.

This instruction will decrement a loop counter and jump to itself while the counter is not zero, simultaneously test the result of the match and jump if the match condition passes, increment the match address memory, and if the loop counts down without a match continue to the next instruction.

A modified version of the loop instruction would increment the jump address for each match check, and could jump to jump table to effect an  $n$ -way branch.  $n$  would only be limited by the size of the loop counter, probably



8 bits (256 count). This would facilitate branching to one of  $n$  PDU types that require different filtering approaches.

The match logic will permit testing fields of 8, 16, or 32 bits wide with a 32 bit mask to ignore certain bits (one mask per match loop, not per match value). These instructions will permit testing multiple fields sequentially ANDing the results, or testing multiple values in a field ORing the results, with one test per clock after two setup instructions.

Figure 5 is a block diagram of the microcode based filter hardware.

## 7.1 Microcode Bit Fields, Instructions, And Timing Estimates

Figure 6 shows the microcode bit fields for the sequencer in the filter hardware. There are three main classes of instructions, loop and jump commands, load start addresses and load loop counter / mask. Figure 7 gives a brief description of each of the micro instructions. Figure 8 contains several sample micro code sequences showing how the micro code can be used to implement efficient loops.

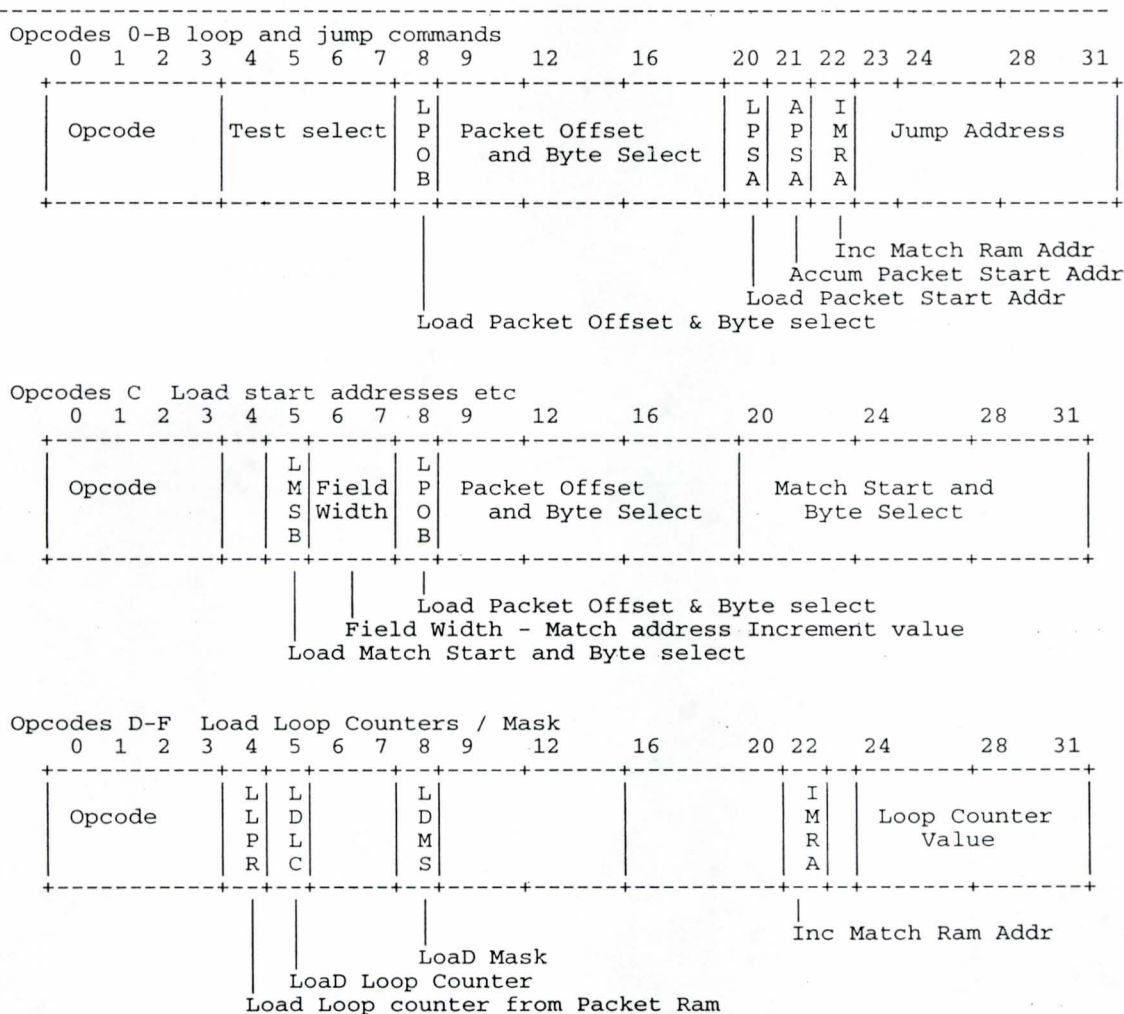


Figure 6: Micro Code Bit Fields

---

OPCODE	Name	Description of instruction
0	JUMP	Jump/Continue on test condition
1,2,3	JDCRx	Jump/Continue on test condition Decrement Loop counter 1 2 3
5,6,7	LOOPx	Three way Jump/Loop/Continue. If Test jump, (PC=Jump Adrs) else if loop cntr X != 0 loop start (PC=Jump_Table_Cntr) else continue (PC++) Decrement Loop counter 1 2 3
8	LDJT	Load Jump Adrs to Jump_Table_Cntr
9,A,B	LOOPxT	Three way Jump_via_Table/Loop/Continue. If Test jump, (PC=Jump_Table_Counter) else if loop cntr X != 0 loop to here (PC=PC) else continue (PC++) Decrement Loop counter 1 2 3 Jump_Table_Counter++
C	LDAD	Load Start Addresses for Match Ram and Packet Offset in Packet Ram
D,E,F	LDLPx	Load Loop Counter 1 2 3 from Loop Counter field or Packet Ram. Optionally load Mask from Match Value Ram

Figure 7: Brief Description of Micro Code Instructions

---

Here is sample micro code to decode PDU types:

```

LDAD   PDU_TYPE    ;Load start addr for Match Value RAM
LDLP1  PDU_SIZE    ;Load loop counter and mask
LDJT   PDU_TBL     ;Load jump table counter
LOOP1T MATCH       ;Loop here while incrementing match value RAM
                     ;also increment jump table counter, and jump to it
                     ;when the PDU type is matched
NOOP    ;Only come here if NO match in PDU type field. Do
         ;some error handling

PDU_TBL:
...
JUMP ESPDU    ;This is a jump table for decoding PDU types
JUMP FIRE
JUMP SIMMAN
...
JUMP XMIT      ;one for each PDU type
JUMP SIGNAL

```

Here is sample micro code test for a match with with a Entity\_id and Radio\_id for a signal PDU. This requires a two instruction loop. The first matches Site and Application, the second matches Entity and Radio id. Only two are needed, because each of the fields are 16 bits, and the hardware can test 32 bits a a time.

```

LDAD  RADIO_ID  ;Load start addr for Match Value RAM
LDLP1 RADIO_SIZE ;Load inner loop counter 1
LDJT  RADIO_LP  ;Load jump table counter to loop start
JMP   RADIO_LP  ;loop starts at RADIO_ID, skip SKIP instruction
SKIP: CONT      ;increment match value RAM and decrement Packet
              ;RAM, don't test last part of id field
RADIO_LP: JMP  NOMATCH SKIP ;no match, skip next test, increment Packet RAM
LOOP1  MATCH KEEP ;Loop to RADIO_ID while incrementing match value
              ;RAM and decrement Packet RAM, Jump to KEEP if
              ;have a match
JMP NO_KEEP     ;fall through if no matches Discard this PDU

```

Figure 8: Sample Micro Code Sequences

Figure 9 is a table with estimated times for execution of the microcode in each of the filter layers. The worst case execution time would be the sum of the four protocol stack layers plus the maximum of the application layer.

If we assume that the Ethernet chip we use as the front end has an internal fifo that holds at least two packets, and the data interface from the Ethernet chip to the filter chip and from the filter chip to the CPU bus, being 32-bit at 33MHz, the time to transmit a packet to or from the filter chip varies from 15 to 250 50MHz clock times. Inputting, filtering, and outputting a max size packet of 1500 bytes takes  $250 + 300 + 250 = 800$  50MHz clocks. The minimum size Ethernet packet takes 51.2 micro seconds, or 2560 50MHz clocks to input. So, worst case, it takes less than one third of the available time. The Packet RAM would not be required to behave as a fifo in this case, with simultaneous input and output.

For 100MHz FDDI or fast Ethernet, we only have 256 50MHz clocks available for the minimum size packet. The Packet RAM would defiantly be required to behave as a fifo in this case as simultaneous input / processing / output would necessary to keep up with the worst case packet rate.

Each loop has over head of 3 instructions plus the size of the loop.

layer	loop size	# of loops	loop size	# of loops	loop size	# of loops	total
Ether	1	3	3	1			18
IP	1	3	2	1			17
UDP	30	1	1	2			41
PDU_h	1	1	1	2			12
PROTOCOL STACK							TOTAL 88
layer	loop size	# of loops	loop size	# of loops	loop size	# of loops	total
Entity State	6x30	1	6	3			210
Fire	2	2					10
Sim Man	2	2					10
Logistics	2	2					10
Designator	2	2					10
Transmitter	6x30	1	1	1			187
Signal	2x30	1	1	1			67
Emission	1	1					4
Receiver	1	1					4
APPLICATION							MAX 210
PROTOCOL STACK + MAX APPLICATION							TOTAL 298

NOTE: the loop sizes that are 30 are list sizes that were some what arbitrarily chosen, and could be changed if needed.

Figure 9: Micro Code Time Estimates for Each Layer



## **8.0 A Typical Dis Exercise**

A typical DIS exercise might consist of several manned helicopter simulators, several manned tank simulators, and several other computer generated tanks. The manned simulators, being quite large and expensive, are likely to be geographically dispersed. The computer generated tanks could be located where ever sufficient computer resources were available. The scenario might be for the helicopters to search for and destroy enemy tanks. Each of the separate simulators needs to know the location of each of the other participants, so that when the other entities are within a certain range, they may be appropriately displayed on the radar screens or out-the-window visual screens. Information on weapons fire and impact is also required to determine if attacks are successful.

Most of the DIS exercises, that have been run to date, have used the broadcast mode of the Ethernet for transmission of the PDUs. This sends all information to all entities and each entity is required to examine each PDU and determine if the data it contains is relevant to its mission. As more entities participate in an exercise and transmit their state data, a greater percentage of the information received by each entity will not be of use to it. This increasing amount of irrelevant data will reduce the performance of both the network and the individual applications that have to process it. It is very desirable to reduce the irrelevant data seen by the network and applications.

## **9.0 Experimental Results**

To test the effectiveness of the filter, several test cases were generated. These were similar to the typical DIS exercise described above.

### **9.1 Rectangular Paths**

The first test case simulated eight tanks each on a rectangular path about 9000 meters by 9000 meters. The different tracks are offset from each other by 5 meters, and the tanks start on different edges: N,E,S,W. The tanks move at about 20-37 miles per hour (each tank is 2.2 mph faster than the preceding tank).

The own ship helicopter travels in rectangular path 7000 meters by 1000 meters, inside the tank paths, about 1000 meters from the tank path edges. The radius that the own ship accepts packets was set to 1400 meters. The own ship moves at about 150 miles per hour. Figure 10 shows three of eight tank path and the own ship path.

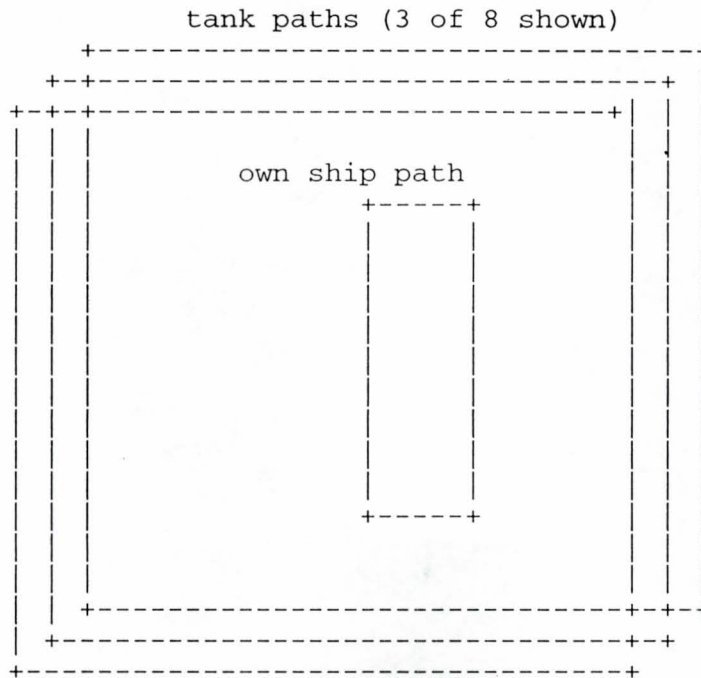


Figure10: Paths of 3 of 8 tanks and own ship path

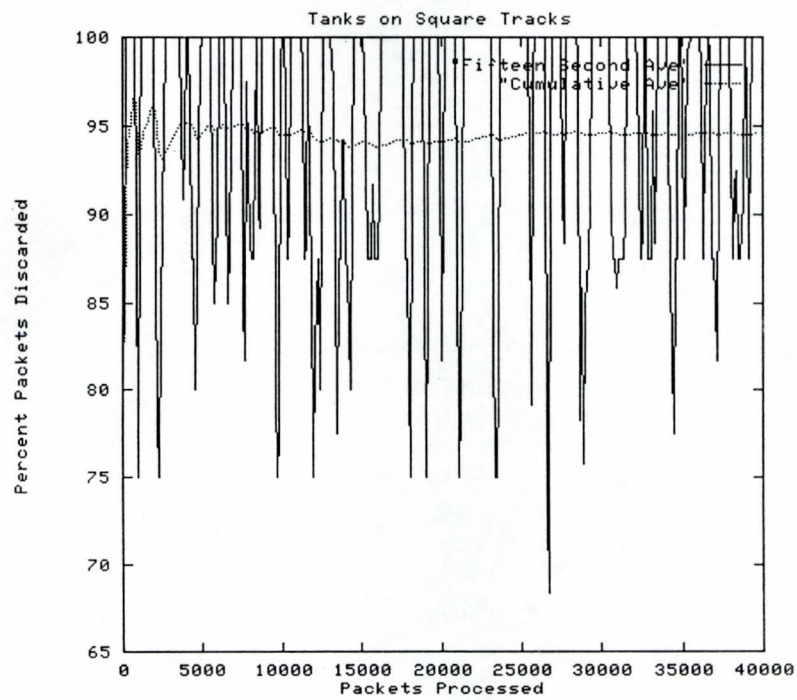


Figure11: Packets Discarded in Rectangular Grid Case

Only Entity State PDUs are simulated here. The tanks and own ship each issue Entity State PDUs once per second. The own ship simulation also issues API calls for the constant data in the other protocol layers once every 12 seconds to prevent automatic timeout after 14 seconds.

The statistics for number of packets processed and bypassed / discarded at each stage of the filter were gathered and figure 11 shows the average/cumulative percentage of packets discarded by the filter (averaged over 15 seconds of simulations time).

## 9.2 Random Path

The previous case is fairly simple and was implemented mostly to debug the filter functions. This case is a more realistic scenario. There are eight tanks traveling in random paths inside the same 9000 by 9000 meter space as the previous case. The tanks maintain their path for a random period between 30 and 60 seconds. At that time, a new  $\partial x$ ,  $\partial y$  and duration are generated. The speeds are still in the 20--40 mph range. The own ship also travels in a random path inside a 7000 by 7000 meter box. Its speed is 100--200 mph. Each tank and own ship produces a PDU packet at a approximately 1 per second. The tanks actual PDU rate is a random number between 0.8 and 1.2 seconds. Figure 12 shows five of the tank tracks, figure 13 shows the own ship path. Figure 14 shows the statistics for the number of packets discarded.

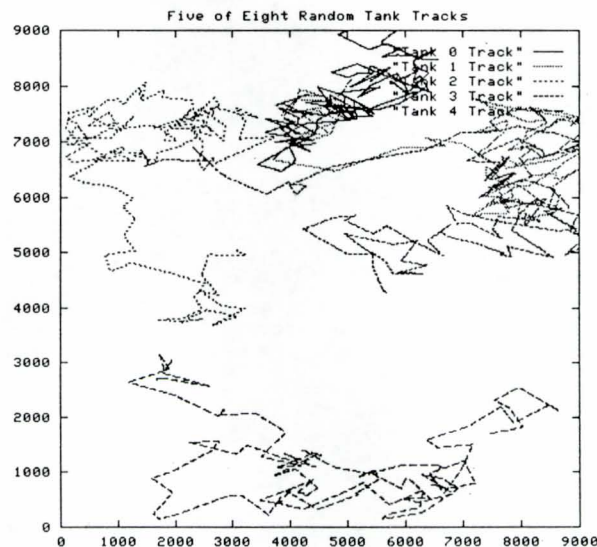


Figure 12: Paths of Tanks in the Random Case

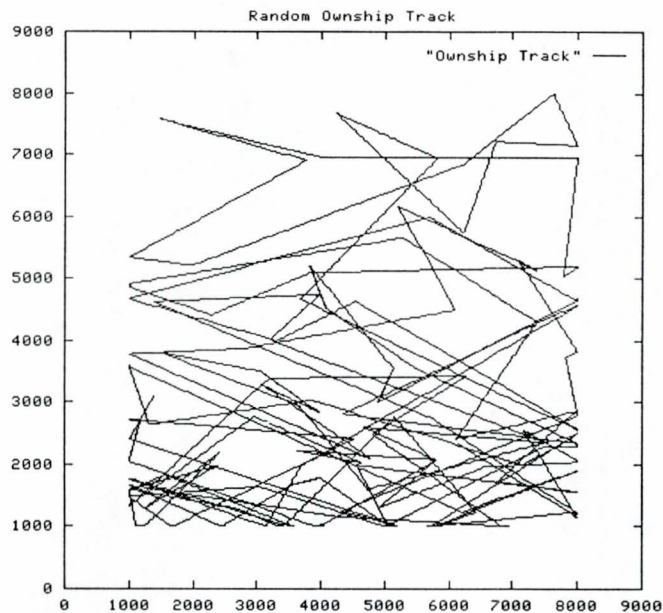


Figure 13: Path of a ownship in the Random Case

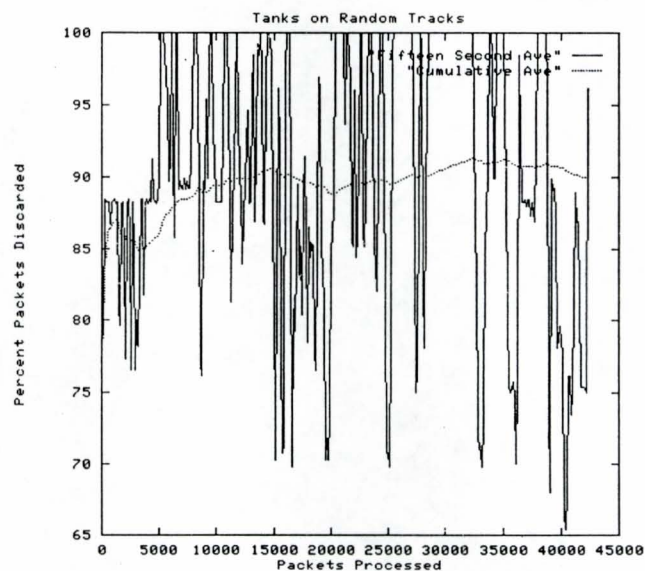


Figure 14: Packets Discarded in Random Case

### 9.3 Actual Simulation Data Replayed

We used some actual data recorded from DIS exercises performed at the I/ITSEC 1994 DIS [7] demonstrations.

Several of the DIS demonstration were examined to find one or two that were similar to the above test cases. Three candidates seemed similar, CGF Demonstration Exercise 5, Helicopter Armed Recon Exercise 9, and Ground Combat Exercise 11. A simple pre-filter was written to extract just the entity state PDUs for one



particular exercise, from the large (16MB files) and write them to smaller files which were run through the filter described here. The Day4-37 file and exercise 9 was selected as the test case.

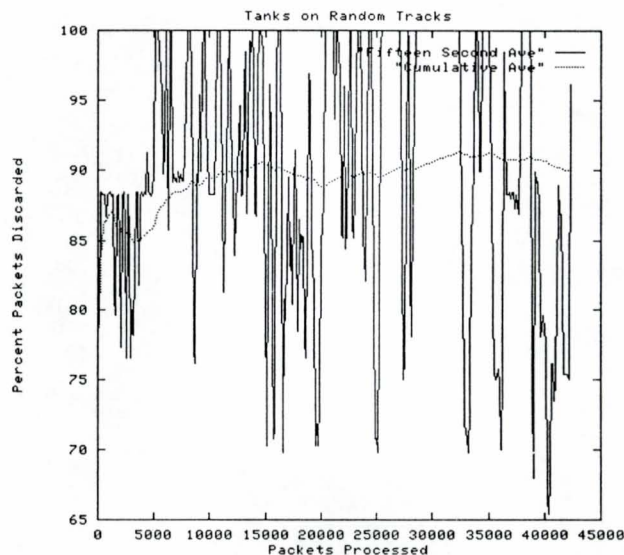


Figure 15: Packets Discarded in iITSEC Data

One of the helicopters was selected as the own ship. The radius that the own ship accepts packets was set to 3000 meters. Each time this a PDU from this entity was read from the file, the filter parameters were updated to reflect its new position. All of the other PDUs were sent through the filter, and statistics were accumulated on the bypass/discard rates.

Another set of 20 test cases were run varying the radius that the own ship accepts packets between 1000-10,000 meters for each of the tank and helicopter as the own ship. There were considerably fewer total packets in these files than the artificial test cases discussed earlier, and the packets per second rate was also slower. The total percentage of packets discarded was also less in this test case. Of course, these discard rates will vary considerably, depending on the exact type of exercise that is being simulated, and how large the gaming area is compared to the area of interest of the particular own ship entity. The statistics for number of packets processed and bypassed /discarded at each stage of the filter are shown in figure 15 (for helicopter as the own ship), figure 16 (for the tank as the own ship) and figure 17 (for various accept ranges).



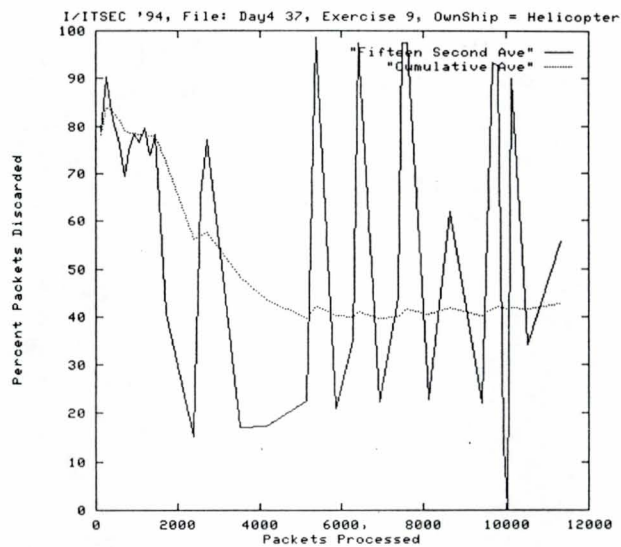


Figure 16: Packets Discarded as a Function of Ownship's Interest

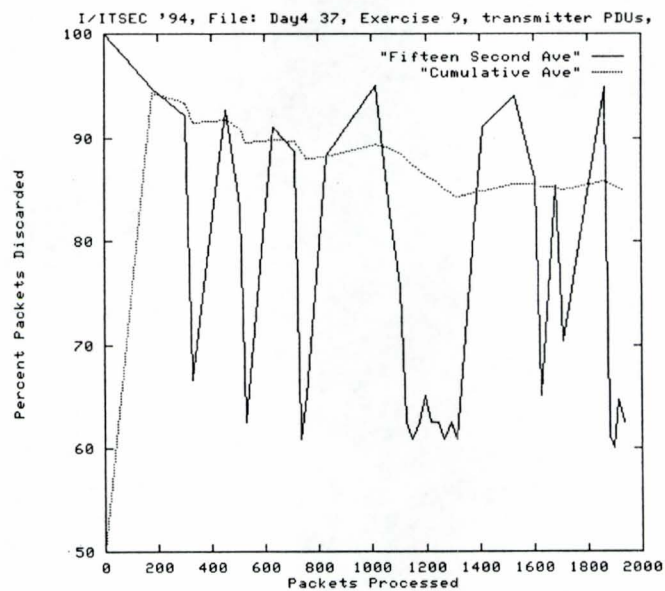


Figure 17: I/ITSEC 94 File: Day4\_37, Exercise 9 Transmitter PDU

## 9.4 Radio Transmissions

Test case four uses the same data file: Day4\_37, as test case three. The raw I/ITSEC data file was pre-processed to extract only the transmitter and signal PDUs. The main test program was modified slightly, to handle the transmitter and signal PDUs. When ever a transmitter PDU was read from the file, it was sent through the filter. If it was bypassed, the filter parameters were updated to accept signal PDUs that matched the entity and radio identifiers from the transmitter PDU. Signal PDUs were just sent through the filter, and bypass/discard statistics maintained.

The Day4\_37 contained transmitter PDUs containing only five different frequencies. Two three cases were run. One selected two of the frequencies, the second selected two other frequencies, and the last selected the final frequency. Oddly, there were only signal PDUs that matched one of the five frequencies. The other four frequencies had no corresponding signal PDUs. The discard statistics are very nearly identical for the first two cases, only one is shown in figure 17. Only transmitter PDUs were bypassed in this case, all of the signal PDUs were discarded, so the discard rates are fairly high. Figure 18, shows the final case. All of the signal PDUs were bypassed here, as well as some of the transmitter PDUs, so the discard rate is fairly low.

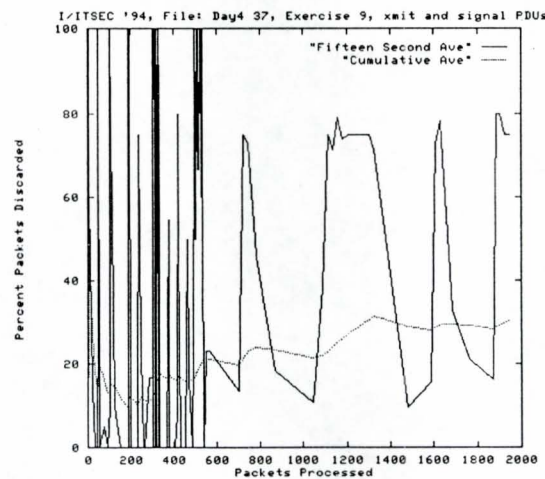


Figure 18: I/ITSEC 94 File: Day4\_37, Exercise 9 Transmitter and Signal PDU

## 10.0 Packet Processing Overhead

Network processing overhead was measured on an actual system and software used at IST. Two sets of test cases were run. The first set used two 80486DX2-66 machines, the second set used two Pentium 75 machines. The two machines were isolated on their own network cable, so only those two machines would generate packets on the Ethernet. The software is the current version of ISTCGF, (Institute for Simulation and Training Computer Generated Forces) This is a simulator that can generate and control a number of entities, as well as display their locations in the data base on screen. One system was setup to send PDU packets, the other only received PDU packets. The exercise numbers were set differently on the two systems. This allowed the receiving system to process the packets through all of the protocol layers and discard them at the application layer with essentially no processing at that layer. The ISTCGF program has built in statistics gathering that reports the total time the simulation was run, the number of loops the executive executed, the number of packets received and discards at each layer of the protocol stack.

For the first test case in each set, the sending host was not running the simulation software, and therefore was not sending any packets on the network. In this baseline case, the simulator was executed on receiving host for one minute. This was timed by a stop watch and the statistics reported typically about a half a second more.

For the other test cases, the sending host was started and two or more tanks were created and set rotating on their axes. This produced about 6.5 packets per second on the network for each pair of tanks. Six more test cases were run with two, four, six, ten, 16, and 24 spinning tanks. Twenty-four entities were as many as the simulator supported. Twenty-four spinning tanks produced about 156 packets per second. One final test case was run with 24 F16 aircraft. This produced about 256 packets per second. A sample main program is shown in Appendix C.

Define the following terms:

- T = Time program is run
- W = Work time per loop
- P = Overhead time per packet
- $P_x$  = No. of packets processed in case x
- $W_0$  = Number of loops processed when there are no packets processed

Then,

$$T = W \times W_0$$

$$T = W \times W_x + P \times P_x$$

If we assume  $T = 1$  second, then:

$$W = 1/W_0$$

$$P = (1 - W_x/W_0) / P_x$$



Case	Seconds	Packets	Loops	Pkts/Sec	Loops/sec	Us/Pkt Ovhd
0 pps	60.858	0	1017838	0.00	16724.80	
2 tanks	60.528	787	1002316	13.00	16559.43	760.46
4 tanks	60.474	1576	992694	26.06	16415.34	709.99
6 tanks	60.583	2373	985239	39.17	16262.53	705.65
10 tanks	60.364	3946	964216	65.37	15973.45	687.23
16 tanks	60.638	6342	941546	104.59	15527.25	684.63
24 tanks	60.638	9490	907692	156.50	14968.97	670.82
24 f16s	60.693	15564	833065	256.44	13725.79	699.26

Table 1: Packet Overhead Experiment Results on a 486DX2-66

Case	Seconds	Packets	Loops	Pkts/Sec	Loops/Sec	Us/Pkt Ovhd
0 pps	60.199	0	266618	0.00	4428.93	
2 tanks	60.254	743	265051	12.33	4398.90	549.88
4 tanks	60.474	1490	264081	24.64	4366.87	568.74
6 tanks	60.144	2227	260786	37.03	4336.02	566.55
10 tanks	60.529	3752	258685	61.99	4273.77	565.20
16 tanks	60.419	6001	252506	99.32	4179.26	567.58
24 tanks	61.023	9125	247555	149.53	4056.74	561.99
24 f16s	61.023	7205	252947	118.07	4145.11	542.77

Table 2: Packet Overhead Experiment Results on a Pentium

From these calculations, the average packet overhead is determined to be 560  $\mu$  s for the Pentium 75 and 702 $\mu$  s for 486DX2-66 machines.

## 11.0 Run Time Overhead

In order to quantify the extra overhead the application has in keeping the filter parameters updated, we profiled the run time of the simulator. It indicates that the application overhead in issuing requests to the filter average about 36 $\mu$  s per second of simulation time (which is very small).

To gauge the amount of extra overhead required to update the hardware filter parameters, run time profiling was performed on Test Case One. The GNU programs: g++ -p -pg -O and gprof were used. 50000 seconds of simulation were performed to obtain meaningful results, as gprof only prints times to 0.01 seconds. This took about five minutes of real time on a 486DX4-100 running under Linux 1.2.8.

The functions whose names begin with `Filter` are the API calls that are being timed. They take 1.79 seconds for 50,000 iterations, which is about 36 microseconds per second of simulation time. The filter chip driver function actually writing data to the filter chip is not in this simulation.

The pertinent results from the gprof output are in Figure 16, (the comments below this table are also from gprof).

% time	self	children	called	name
1.3	0.50	1.99	50001/50001	main
	0.50	1.99	50001	Own ship(int)
	1.64	0.00	50001/50001	FilterOwnShipPosition()
	0.20	0.00	50000/450000	MoveSquare()
	0.06	0.00	4167/4167	FilterEntityStateRangeSQ()
	0.05	0.00	4168/4168	FilterUDP-Port()
	0.01	0.00	4167/4167	FilterEthernetAddress()
	0.01	0.00	4167/4167	FilterIP-Address()
	0.01	0.00	4167/4167	FilterUDP-Mode()
	0.01	0.00	4167/4167	FilterPDU-Header()

Table 3: Gprof Results of Test Case One

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

## 12.0 Conclusions

There is compelling evidence that filtering packets at the Ethernet level would enable low-cost workstations to spend more of processing time on useful things, such as better visual effects, and not missing needed packets. On the system timed, a 486DX2-66, merely receiving and discarding 256 entity state PDU packets per second, used 18% of the total cpu time. The system was found (by others) to be able to process about 100 packets per second before packets start being dropped. If a hardware filter was available to offload some of the filtering tasks, the cpu could process more needed packets, rather than arbitrarily discarding useful and well as unneeded packets, when it ran out of processing time.

We also provided a well-defined API interface to the filter. The filter itself is simple, and involves no floating point computations and can be easily integrated into existing Ethernet interface cards.

## 13.0 Acknowledgments

This work is funded by US Army STRICOM under the contract number N61339-94-C-0024 as a part of TRIDIS project. The first author is also partially supported by the US Army Research Office under the contract number DAAH04-95-1-0250 as a part of Advanced Distributed Simulation Research Consortium (ADSRC). However, the views and conclusions drawn herein are those of the authors and do not necessarily reflect the position or the policy of the federal government, any of the sponsors of this work or the University of Central Florida.



## 14.0 References

- [1] IEEE, "IEEE Standard 1278.1, Standard for Distributed Interactive Simulation -- Application Protocols," Institute of Electrical and Electronics Engineers, Inc., 1995.
- [2] M. Bassiouni, M. Chiu, Jim Williams. "Improving the Reliability of Relevance Filtering," 13th Workshop on Standards for the Interoperability of Distributed Simulations, pages 331-336, September 1995.
- [3] Robert Kerr, Christopher Dobosz. "Reduction of PDU Filtering Time Via Multiple UDP Ports," 13th Workshop on Standards for the Interoperability of Distributed Simulations, pages 343-349, September 1995.
- [4] Keven L. Russo, Lawrence C. Schuette, Joshua E. Smith, Matthew E. McGuire. "Effectiveness of Various New Bandwidth Reduction Techniques," 13th Workshop on Standards for the Interoperability of Distributed Simulations, pages 587-591, September 1995.
- [5] J. Mark Pullen, Elizabeth L. White. "Analysis of Dual-Mode Multicast for Large Scale DIS Exercises," 13th Workshop on Standards for the Interoperability of Distributed Simulations, pages 613-621, September 1995.
- [6] Daniel J. Van Hook, Steven J. Rak, James O. Calvin. "Approaches to Relevance Filtering," 11th Workshop on Standards for the Interoperability of Distributed Simulation, pages 367-369, September 1994.
- [7] Sandra E. Cheung. "Analysis of Network Traffic from the I/ITSEC 1994 DIS Interoperability Demonstrations," Technical Report IST-TR-95-09, Institute for Simulation and Training, University of Central Florida, April 1995.
- [8] W. Richard Stevens. "TCP/IP Illustrated", volume 1, Addison Wesley, 1994.

## APPENDICES

### A. API Function Prototypes For Stages 1, 2, 3

```
/* api.h typedefs and function prototypes for API calls */

/* Some obvious typedefs */
typedef int BOOL;
typedef unsigned char UINT_8;
typedef unsigned short UINT_16;
typedef unsigned long UINT_32;
typedef UINT_32 UINT_64[2];
typedef float FLOAT_32;
typedef double FLOAT_64;

typedef enum {
    Filter_None = 0, /* Filter vs Bypass modes */
    /* Bypass all, filter is off - default */
    /*
    Filter_Self, /* Bypass multicast and Broadcast */
    Filter_Multicast, /* Bypass Broadcast and self */
    Filter_Multi_Self, /* Bypass Broadcast */
    Filter_Broadcast, /* Bypass multicast and self */
    Filter_Broad_Self, /* Bypass multicast */
    Filter_Multi_Broad, /* Bypass only self */
    Filter_All /* Bypass none */
} FILTER_ETHERNET;

int FilterEthernetAddress(FILTER_ETHERNET Mode); /* Filter vs Bypass
modes */

typedef enum {
    Discard_None = 0, /* Discard vs Filter modes */
    Discard_Broadcast, /* Filter all - default */
    Discard_Multicast, /* Filter multicast */
    Discard_Multi_Broad, /* Filter Broadcast */
    /* Filter only self */
} DISCARD_IP;

int FilterIP_Address(DISCARD_IP Mode); /* Discard vs Filter modes */

typedef enum {
    Add = 0, /* Add/Delete/Update select */
    Delete, /* Add entries in List that follows */
    Update, /* Delete entries in List */
    /* Update entries in List */
} ADD_DELETE;

int FilterUDP_Port(ADD_DELETE Select, /* Add/Delete/Update list of PDU
ports */

    int *Start, /* Start index of list
For ADD, returns actual start index

*/

    int *Size, /* Size of Filter Array: List
Returns actual number accepted */
    UINT_16 *List); /* array of ports PDUs are on */
```

```

typedef enum {
    BypassPDU_BypassOther = 0, /* Bypass PDU's and Bypass Other packets
*/
    BypassPDU_DiscardOther, /* Bypass PDU's and Discard Other packets
*/
    FilterPDU_BypassOther, /* Filter PDU's and Bypass Other packets
*/
    FilterPDU_DiscardOther /* Filter PDU's and Discard Other packets
*/
} PDU_OTHER;

int FilterUDP_Mode(PDU_OTHER Mode); /* select Bypass/Filter PDU's/Other
mode */

```

## B. API Function Prototypes For The Application Stage Of The Filter

/\* Note: Generally, the Entity Site and Application are the same in all of the following PDU types: all Logistics, all Simulation Management Fire, and Designator. \*/

```

int FilterPDU_Header(UINT_8 Version, /* Protocol version to filter */
                    UINT_8 ExerciseID); /* Exercise ID filter */

```

```

int FilterLogisticsEntitySite(BOOL All, /* Bool, accept all entities ID
*/
                             UINT_16 EntitySite, /* Entity Site to
bypass */
                             UINT_16 Application); /* Application to
bypass */

```

```

int FilterSimManEntitySite(BOOL All, /* Bool, accept all entities ID */
                          UINT_16 EntitySite, /* Entity Site to bypass
*/
                          UINT_16 Application); /* Application to bypass
*/

```

```

int FilterFireEntitySite(BOOL Unknown, /* Bool, accept unknown entity ID
*/
                        UINT_16 EntitySite, /* Entity Site to bypass */
                        UINT_16 Application); /* Application to bypass
*/

```

```

int FilterDesignatorEntitySite(BOOL Unknown, /* accept unknown entity ID
*/
                              UINT_16 EntitySite, /* Entity Site to
bypass */
                              UINT_16 Application); /* Application to
bypass*/

```

```

typedef struct {
    FLOAT_32 x; /* 32 bit IEEE floating point */

```



```

    FLOAT_32    y;
    FLOAT_32    z;
} OWN_SHIP_POSITION;

int FilterOwnShipPosition(ADD_DELETE Select, /* Add/Delete/Update list
*/
                          int Index, /* Index of OwnShip Entity (0-31)
*/
                          OWN_SHIP_POSITION *OwnShipPosition);

typedef enum              /* Speed range for entities */
{
    SPEED_OTHER,          /* Other */
    FIXED,                /* Culture and Environmental */
    SLOW,                  /* Life Forms */
    MEDIUM,               /* Tanks, Trucks, Boats */
    FAST,                  /* Helicopters */
    VERY_FAST,            /* Fixed Wing Aircraft */
    HYPER_FAST            /* Munitions */
} KIND_DOMAIN_SPEED;

int FilterEntityStateRangeSQ(ADD_DELETE Select, /* Add/Delete/Update list
*/
                             int OwnShip, /* OwnShip index to apply these
to */
                             int *Start, /* Start index of list
For ADD, returns actual start
index
This is one of
KIND_DOMAIN_SPEED */
                             int *Size, /* Size of RANGE Array: List
Returns actual number accepted
*/
                             FLOAT_32 *List);

typedef struct {
    UINT_64 HighFreq;      /* High Freq limit */
    UINT_64 LowFreq;       /* Low Freq limit */
    FLOAT_32 HighBandwidth; /* High Bandwidth limit */
    FLOAT_32 LowBandwidth; /* Low Bandwidth limit */
} FREQ_BANDWIDTH;

int FilterTransmitter(ADD_DELETE Select, /* Add/Delete/Update list */
                      int *Start, /* Start index of list
For ADD, returns actual start index
*/
                      int *Size, /* Size of FREQ_BANDWIDTH Array: List
Returns actual number accepted */
                      FREQ_BANDWIDTH *List);

typedef struct {
    UINT_16 EntitySite;    /* Entity Sites to bypass */
    UINT_16 Application;   /* Applications to bypass */
    UINT_16 EntityID;      /* Entity IDs bypass */
    UINT_16 RadioID;       /* Radio IDs bypass */
} RADIO_ID;

int FilterSignal(ADD_DELETE Select, /* Add/Delete/Update list */
                 int *Start, /* Start index of list

```

```

/*
                                For ADD, returns actual start index
int *Size,          /* Size of RADIO_IDs Array: List
                    Returns actual number accepted */
RADIO_ID *List);

int FilterEmission(BOOL discard); /* Bool, discard all emission PDUs if
set */

int FilterReceiver(BOOL discard); /* Bool, Discard all receiver PDUs if
set */

```

### C. Main Loop Code Of ISTCGF

```

for (;;) {
    rightnow = GetTime(); /* Update the current time */
    MathOK(); /* No math errors so far? */
    loop_cnt++; /* Count this go-around */
    CheckNetInterface(); /* Check for incoming network packets */
    MsgScan(); /* Dispatch highest priority message */
    SetCCB(cons_mgr_cb);
    ConsoleCheck(rightnow);
    CheckGraphicsMsg();
}

```

### D. API Function Calls Code

For the purposes of simulating the functionality of the filter functions, the actual filter parameters were stored in C memory structures. These API function calls, which are called by the receiving DIS simulator, write these structures, and the filter functions read them.

In an actual hardware implementation of the filter, each of these API calls would have a few lines of additional code to write the data structure to the filter chip. If the filter memory is memory-mapped in the CPU memory space, this could be as simple as declaring and mapping the structures in the actual memory space of the filter. In this case, API functions would not have to be changed at all. Alternately, a memcpy() call may be used. If the filter memory is I/O port mapped to the CPU, a DMA request would be required.

0000082