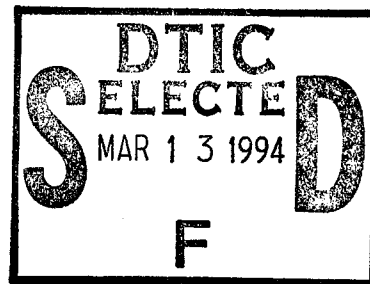


NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

SOFTWARE TESTING TOOLKIT FOR DISTRIBUTED SIMULATIONS

by

Mitchell K R Turner

September 1994

Thesis Advisor:
Co-Advisor:

Gary Porter
Michael Macedonia

Approved for public release; distribution is unlimited.

19950308 160

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1994		3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE SOFTWARE TESTING TOOLKIT FOR DISTRIBUTED SIMULATIONS (U)				5. FUNDING NUMBERS	
6. AUTHOR(S) Turner, Mitchell K R					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE A	
13. ABSTRACT (Maximum 200 words) This thesis discusses the need for and design of a software toolkit to monitor Distributed Interactive Simulation (DIS) network performance. Plans to merge virtual environment and wargaming simulations into combined exercises will have significant performance effects on existing networks, but the tools to quantify the impact are lacking. Given the need for performance measurement tools, the network environment is described and two software development methods, Motif and Tcl/Tk, are considered. The merits of Tcl/Tk, including extensibility to access DIS networks and ease of application development, resulted in a programming environment well-suited for this requirement. The network toolkit design is presented, including the required modifications to Tcl/Tk. Areas for future research include using the PDU Monitor and the Tcl/Tk application, and expanding their capabilities.					
14. SUBJECT TERMS Virtual reality, networked virtual worlds, distributed interactive simulation, DIS, Tcl, Tk, network performance, stripchart				15. NUMBER OF PAGES 80	
				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		

Approved for public release; distribution is unlimited.

Software Testing Toolkit for Distributed Simulations

by

Mitchell K R Turner
Lieutenant, United States Navy
B.S., United States Naval Academy, 1986

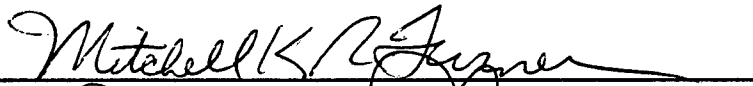
Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY
(Space Systems Operations)

from the


NAVAL POSTGRADUATE SCHOOL
September 1994

Author:




Mitchell K R Turner

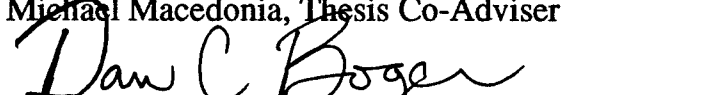
Approved by:



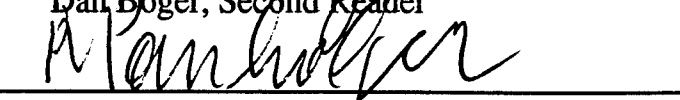
Gary Porter, Thesis Advisor



Michael Macedonia, Thesis Co-Adviser



Dan Boger, Second Reader



Rudolf Panholzer, Chairman
Space Systems Academic Group

ABSTRACT

This thesis discusses the need for and design of a software toolkit to monitor Distributed Interactive Simulation (DIS) network performance. Plans to merge virtual environment and wargaming simulations into combined exercises will have significant performance effects on existing networks, but the tools to quantify the impact are lacking.

Given the need for performance measurement tools, the network environment is described and two software development methods, Motif and Tcl/Tk, are considered. The merits of Tcl/Tk, including extensibility to access DIS networks and ease of application development, resulted in a programming environment well-suited for this requirement.

The network toolkit design is presented, including the required modifications to Tcl/Tk. Areas for future research include using the PDU Monitor and the Tcl/Tk application, and expanding their capabilities.

Accession For	
NTIS	CRA&I <input checked="checked" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	BACKGROUND	3
	A. DISTRIBUTED SIMULATIONS	3
	B. NATIONAL SYSTEMS AND WARGAME SIMULATIONS	4
	1. National Systems	4
	2. Wargame Simulations.....	5
	3. Janus Wargame Example.....	5
	C. NEED FOR NETWORK MONITORING TOOLS.....	6
III.	DIS PROTOCOL AND NETWORK ENVIRONMENT	8
	A. DIS PROTOCOL	8
	B. NETWORK LIBRARY	9
	C. SIMULATION NETWORK ARCHITECTURE	10
IV.	SOFTWARE DEVELOPMENT ENVIRONMENT OPTIONS	11
	A. MOTIF AND C.....	11
	B. TCL/TK.....	13
	C. BEST ALTERNATIVE	13
V.	IMPLEMENTATION AND EVALUATION	15
	A. TOOLKIT DESIGN.....	15
	1. Tk Extensions	15
	a. netAppInit.c.....	16
	b. disnetlib.c	16
	2. PDU Monitor Design	16
	3. PDU Monitor Features.....	18
VI.	TEST AND EVALUATION	21
	A. INPUT ERROR DETECTION	21
	B. EXCEPTION HANDLING	23

C. PDU MONITOR PERFORMANCE	23
VII. FUTURE RESEARCH	27
A. NETWISH.....	27
1. Using netwish	27
2. Extending netwish	27
B. USING THE PDU MONITOR	28
C. MODIFYING THE PDU MONITOR	28
VIII. CONCLUSIONS	30
APPENDIX A. C CODE TO ADD DIS NETWORK FUNCTIONS TO TK.....	31
APPENDIX B. C FUNCTIONS TO IMPLEMENT DIS NETWORK ACCESS	34
APPENDIX C. TCL SCRIPTS IMPLEMENTING THE DIS PDU MONITOR.....	41
APPENDIX D. INSTALLATION INSTRUCTIONS	70
LIST OF REFERENCES	73
INITIAL DISTRIBUTION LIST	74

I. INTRODUCTION

Various warfare and support communities in all of the military services are building and using computer simulations to test equipment designs and tactical scenarios. While many of these efforts have been undertaken without consideration of other ongoing efforts, there are projects sponsored by several agencies, most notably the Advanced Research Projects Agency (ARPA), to conduct simulations on a computer network and allow the various wargames and simulators to interact with each other in a "virtual world." Concurrently, managers of U. S. national systems are developing simulators to function in a training and exercise role. Incorporation of national systems' data would enhance the realism of the virtual world and model the interaction between users and the national systems. (Reddy, 1992, pp. 10-14)

The background for this research centers on the joining of two distinct types of simulations in distributed networks exercises. The first is a *system simulation*, or *entity simulator*, which refers to computer systems which model operator control with graphical displays on a monitor or cockpit simulators which are full-scale models of the system of concern. A single vehicle entity is modeled, and its actions are directly controlled by an operator in real time, often by input devices such as joysticks. The second simulation type is the *exercise simulation* or *wargame simulator*, referring to computer programs which model the interaction between forces such as army divisions or naval battle groups. The forces modeled by these wargames are controlled by operators representing force command staffs. Control instructions, which may or may not be in real time, are input by keyboard commands, and results of interaction between entities are determined by programmed probabilistic relationships. Both types of simulations model a virtual world in which the forces or entities operate. Distributed simulations use a network to bring entities represented by different computers into the same virtual world. Initial distributed simulation efforts focused on entity simulators, but further efforts such as the ARPA

project mentioned above attempt to merge the entity and wargame simulators in the same virtual space.

As the scope of such virtual environment exercises has been broadened, the increased numbers of players has begun to result in significant effects on network performance. Knowledge of the impact of larger numbers of participants is limited since overall control of joint efforts is dispersed among several agencies and network managers.

All players in network simulations must use the same simulation protocol, and the Distributed Interactive Simulation (DIS) protocol has become the standard. The DIS protocol defines standards for application level communication among independently developed simulators. Entities in the virtual world communicate their status and activities to other entities via protocol data units (PDU) structured according to the DIS protocol. (Macedonia, 1994, p. 3)

Network monitoring tools are often available for the operating systems used with the various computers used to run simulations, but these tools deal only with bulk network traffic, which may or may not be related to the ongoing simulations. Since the different types of DIS traffic are not monitored, the impact of simulation participants and their different activities remains unknown.

The main thrust of this research has been to develop an extensible tool kit that can be used while distributed simulations are being run and will provide real-time indications of the network load resulting from increased numbers of players and their specific activities in the virtual world. This thesis will also present background on the simulation efforts currently underway, background for the need for a network testing tool kit, and an investigation of a new software development environment.

II. BACKGROUND

The military services are relying on simulations to a greater extent than ever before, as resources to provide live training and exercises have become severely limited. Computer technology has provided the means to represent realistic operational conditions without the cost or risk to personnel and equipment of live operations in the field. Computer-based exercises provide the advantages of precise real-time and post-exercise analysis and the ability to create or re-create very specific scenarios, including those simulating lethal environments. (Reddy, 1992, pp. 2-5)

Military computer simulations can be divided into two broad categories – those used for training and exercise support (wargames) and those used for weapons system development and testing (entity simulations). The scale of simulators can range from representing one person or vehicle (an entity), such as in a cockpit simulator, to a theater-level wargame involving Joint Task Force staffs (Reddy, 1992, p. 10). The integration of multiple entities and simulation types is the underlying idea for distributed simulations.

A. DISTRIBUTED SIMULATIONS

Though the technological advances in simulation implementations have provided increased realism, the benefits have typically been limited to the specific individual, unit, or staff using a particular system. The advent of world-wide, high-speed data networks and distributed software has allowed the possibility of those entities interacting with each other as if they were in the same operational area, while using simulators in multiple locations scattered across several time zones. The Naval Postgraduate School (NPS) has implemented a three-dimensional virtual environment called NPSNET-IV, which allows users on a network to sense and interact with each other on common, realistic, and dynamic terrain. To the users it appears as if they are operating in the same space, time, and terrain. Dynamic changes to the distributed simulation such as explosion craters and

structural modifications appear the same to each player. This is the basis for the concept of a "virtual world." (Macedonia, 1994, p. 5ff)

The sharing of the space within a virtual world allows the addition of different types of simulators as well as systems in different physical locations. As long as a system has a means of sensing the characteristics of the virtual world and can input changes to it (the protocol for which is addressed in the next chapter), the types of entities that can be represented are unlimited. A thrust of current ARPA activity is to allow graphic workstations, advanced entity simulators (e.g., aircraft cockpit simulators), and even actual operational equipment to participate in the representation of a computer-generated virtual world (Reddy, 1992, p. 14).

A significant aspect of this type of simulation is that it affords players the realism of dynamic and unpredictable interaction with each other, as opposed to constructive simulations where the results of encounters are determined by pre-programmed probabilities of detection, impact, or kill. As in the real world, outcomes can be determined by user skill, reaction time, and decision-making ability.

B. NATIONAL SYSTEMS AND WARGAME SIMULATIONS

As NPSNET has been developed to implement a virtual world which is based on actual terrain characteristics and in which simulated entities can freely move and interact with other entities, wargame simulations are being developed to use DIS and will increase the use of the same network resources.

1. National Systems

For example, national systems managers have been building computer systems, for the purpose of user training and wargaming, to model the use of the national systems. Some examples follow: (OSO, 1993, pp. 19-50)

- Synthetic Imagery Generation System (SIGS) is designed to provide simulated national level imagery for military exercises by modeling a visible wavelength sensor and providing gray-scale photo simulations. Scenes can be built which reflect atmospheric or daylight conditions.
- Stand-Alone TENCAP Simulator (SATS) is a PC-based system which can generate and display user-produced platform tracks and ELINT contacts. SATS can generate

messages readable by TRAP (TRE (Tactical Receive Equipment) And Related Applications) broadcast processors.

- Exercise Capability (EXCAP) is an intelligence data simulator that can provide information in a variety of message formats to support tactical exercises, operational tests, and training events. EXCAP models national and theater collection systems, and provides tools to build exercise scenarios and perform collection planning and tasking.
- National Wargaming System (NWARS) is an intelligence asset simulator which supports training and exercises. It models tasking, collection, and reporting of national systems while interfacing with various scenario generators and other simulators. NWARS reports can be distributed by operational command, control, and communications systems.

2. Wargame Simulations

The purpose of intelligence-based wargame simulations is to model the performance of various intelligence collection and reporting systems. The characteristics and many of the actions of the entities which are being monitored by the simulated national systems are planned in advance and incorporated into programmed scenarios. The possibility of real-time modification of the terrain and reaction to other players' actions is very limited.

If simulators of both types – individual entity and wargame – could operate in the same distributed architecture, their strengths could be combined to enhance the realism of both. The virtual world would provide a realistic and dynamic scenario for the national systems models to monitor, while the intelligence picture generated could be used by the entities in the virtual world to operate as they would in a live tactical environment.

3. Janus Wargame Example

The Janus wargame simulator is an example of current efforts to merge DIS and constructive simulations. Janus is a two-sided ground combat simulation of engagements up to the brigade versus division level. It runs on one machine and displays opposing forces on several. Controllers at each workstation interactively direct actions of their units. Engagements between individual land-based fighting systems are stochastic, i.e., the results of the engagements depend on probability. (Johnson, M., 1994, p. 4)

Work at the Naval Postgraduate School has resulted in the development of the World Modeler, which translates the DIS protocol so Janus can send and receive PDUs on a DIS network (Johnson, M., 1994, p. 9). Since Janus models the actions of forces, the number of entities in a distributed simulation is significantly increased when Janus is merged with individual entity simulators. The specific impact that Janus would have on the network was unknown as the World Modeler was built, demonstrating the need for tools to obtain a clearer picture of network activity.

C. NEED FOR NETWORK MONITORING TOOLS

Current distributed simulations are capable of generating significant network traffic. Network traffic volume is directly related to the number of entities in a simulation and how often changes in their states must be broadcast to other participants. Knowledge about the impact of increasing numbers of entities operating in the virtual world has been limited to real-time indications of bulk network traffic or post-exercise analysis of more specific data. These two observation methods have significant drawbacks. Measuring the total network load gives little or no indication of the traffic volume related only to the simulation. After-the-fact analysis provides a picture of the load as a function of time, but this picture is isolated from the many other variables present when the measurements were taken.

With the exception of post-exercise analysis of PDU rates (Macedonia, 1994, p. 12), statistics concerning the traffic loads from current DIS simulation efforts are not available. Visits to the National Test Facility (Colorado Springs) and the Naval Research Laboratory (Washington) revealed that data collection has been limited to bandwidth utilization statistics for all network activity. Even less understood is the future network load resulting from the participation of emerging wargame systems such as the national systems simulators described earlier. The traffic resulting from modeling the observation and reporting of actions of all the entities (in addition to the reporting of actions, status, and location by the entities themselves) has not been quantified, leaving as an uncertainty

the impact on the current network architecture and limiting the ability to define future requirements.

The integration of the Janus system with DIS described earlier is an excellent example of the current lack of understanding of the network traffic loads caused by merging new wargame and entity simulations in a single DIS network. Observation of the World Modeler development revealed that the integration of Janus into DIS resulted in unexpected network impacts. The World Modeler programmer used a prototype version of the PDU monitor developed in this research to ascertain and mitigate the impact of the Janus system on the DIS network.

Merging the activity of disparate systems into a single distributed simulation has necessitated the development of a common means of sharing data between participants and representing the characteristics of the virtual world. The DIS protocol provides this common link, as well as the basis for tools to monitor the impact of the simulations on the data network.

A network monitoring toolkit must be designed with the DIS protocol as its foundation, to separate the traffic generated by the various uses of the computer network, such as file transfers and text, voice, and video messaging, and the traffic generated by distributed simulations. This will allow the effects of the DIS-related activities to be isolated from other network activities. A limited tool has been built at NPS to count DIS data packets and collect the volume counts in text files (Macedonia, 1994, pp. 12-13), but use of the data is delayed hours or days until the exercise is completed or at least frozen. Then the data must be transformed into a format, preferably graphical, which can be used by analysts. When the results are available, they are divorced from the events occurring when the data was collected.

A monitoring tool built on DIS protocols should present its results using a real-time graphical display of the DIS network environment, which will aid in traffic analysis and troubleshooting as the simulation runs. Development and use of such a tool is the topic of this thesis and will be described in greater depth in the chapters that follow.

III. DIS PROTOCOL AND NETWORK ENVIRONMENT

To observe the effects of DIS-related activities which occur during distributed simulations, a network performance measurement toolkit must interface with the specific protocol used for communication between hosts. The following paragraphs discuss the relationship between the network toolkit developed for this research and the DIS network environment.

A. DIS PROTOCOL

Version 2.03 of the Distributed Interactive Simulation (DIS 2.03) protocol defines standards for application level communication among independently developed simulators (e.g., full-scale cockpit simulators, computer models, and instrumented live vehicles operating in the field) (Macedonia, 1994, p. 3). NPSNET-IV uses DIS 2.03 protocol data units (PDU) to communicate information between entities. The PDUs are sent on the network using Internet Protocol (IP) Multicast, which allows the IP packets to be sent to defined groups of hosts instead of being broadcast to all hosts.

The DIS PDUs convey information about individual entities, events, and the state of the simulation exercise. There are 27 types of PDUs which are used to carry information about entity states (location and movement), activities (e.g., weapons firing or munitions detonation), supporting actions (re-supply and repair), and simulation control (Institute, 1993). The details of the structure of each PDU type are fully explained in the Institute for Simulation and Training (1993) document; some representative examples of the various PDU types are presented below:

- Entity State (type 1) – contains type of entity, location, course, speed, altitude, flag, markings, and capabilities.
- Fire (type 2) – information about source, type, and movement of fired munitions.
- Repair Complete (type 9) – report from damaged entity after repair operations.
- Stop Freeze (type 14) – signal to freeze the distributed simulation exercise.

B. NETWORK LIBRARY

The network toolkit must read PDUs as they arrive on the network and obtain relevant information from the IP packet header or the PDU structure; for the purpose of this research, the PDU type and sending host are required. NPSNET-IV includes a library of C language functions to directly access the DIS network. The same library functions are used by all applications developed for use on NPSNET. These functions were incorporated into the network toolkit, which means that changes in the network code will not require major software modifications of the toolkit. The application can simply be recompiled with the new library functions.

The monitoring process begins with a call to the library function *net_open*. This function establishes a connection with a DIS network using a specified network port, multicast group address, and exercise identification. *Net_open* spawns a process (called *receiveprocess*) which continually monitors the network for incoming PDUs and copies them to a queue (called the arena). Application software, such as the PDU Monitor developed in this research, uses the *net_read* function to read PDUs from the queue.

Figure 1 shows the interaction of *receiveprocess* and *net_read*. The *net_read* function

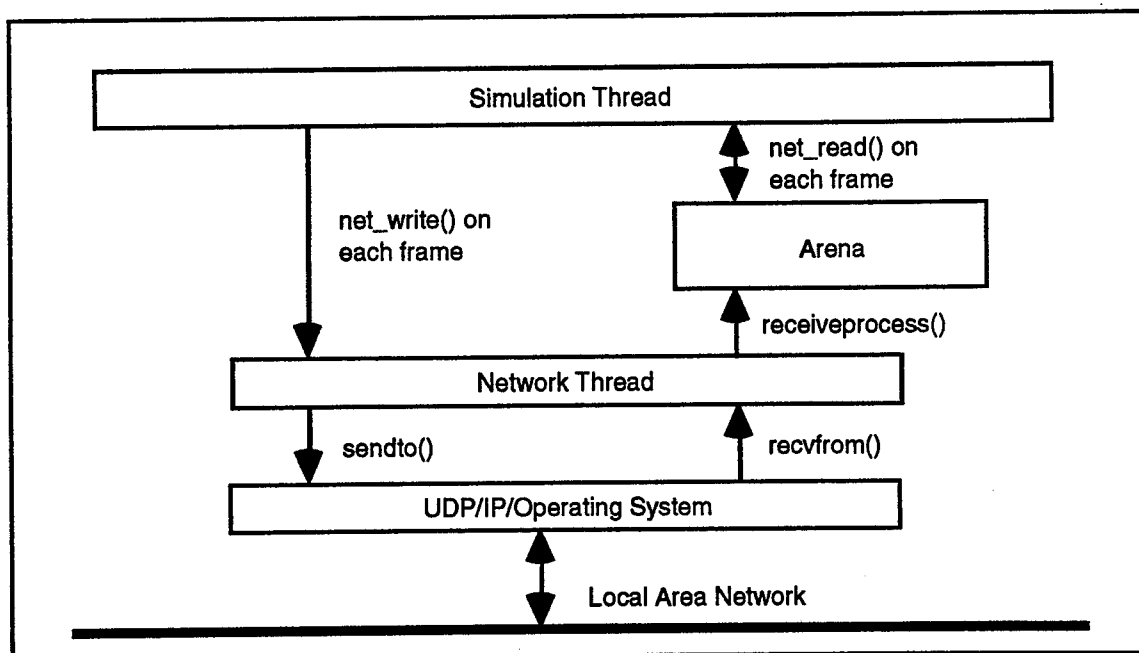


Figure 1. NPSNET-IV Network Library Functions (Macedonia, 1994, p. 20)

returns the number of pending PDUs in the queue, the data in the structure of the next PDU, the PDU type (as an integer value), and address information from the IP packet header. The advantage of having separate receiveprocess and net_read functions is that receiveprocess runs constantly with the sole purpose of ensuring no arriving data is lost, while net_read runs at either user-specified intervals or in between the application's other activities (such as screen updates or calculations). However, net_read must still be called often enough to prevent the queue from filling up, or PDUs will be lost. (Macedonia, 1994, p. 20)

C. SIMULATION NETWORK ARCHITECTURE

DIS 2.03 simulations can be run in a broadcast or multicast mode. In a broadcast mode, all hosts on the network receive all PDUs. This mode can be used in a Local Area Network (LAN) or private network environment, since broadcasting all exercise data to every host would impact almost every system. The multicast mode allows hosts not subscribing to a defined group to automatically reject packets not destined to that computer.

The multicast mode allows existing internetworks and commercial services to be used for distributed simulations, rather than relying on private networks. Since the IP Multicast protocol handles the delivery of packets, the DIS protocol is independent of the topologies and technologies of the local and long-haul services. (Macedonia, 1994, p. 15)

IV. SOFTWARE DEVELOPMENT ENVIRONMENT OPTIONS

Several considerations were examined in choosing a software development environment to develop a PDU monitoring tool. The program must be able to access the DIS network library functions, which are written in C. The resulting application must be able to display information in a windows environment, read the network queues, perform simple calculations, and update displayed graphs quickly enough that the network queue does not fill up and overflow. The programming language should be conducive to future modifications of the monitoring tool as well as provide the flexibility for other uses in the network environment. Two options were considered, the first being the standard method for programming in the graphical Xwindows system, the second a new toolkit developed for rapid development and uncomplicated testing and modification. Included in the description of each method is a small program which creates a button, assigns to it the text "Hello world from UNIX REVIEW," and gives the button the functionality to quit the program (Johnson, E., 1994, pp. 87-90). These short programs are included as examples of the difference in programming complexity between languages being considered.

A. MOTIF AND C

The standard for the graphical window interface commonly used on computer workstations is X, a protocol designed for networks and usually implemented in C. *Motif* is a window manager which controls the display of widgets (the components of windows), which include buttons, menu bars, and text entry boxes, in accordance with the X protocol standard. Like X, Motif is a library of C programs that can be used in applications, resulting in pre-programmed widgets that are consistent with X. (Ferguson, 1994, pp. 6-10)

Programs written in Motif/C are complex, take a considerable time to compile or recompile after a change, and require a solid understanding of both C and Motif to

develop or modify. However, since the resulting application is compiled and run as a binary program, the speed of a program developed in Motif/C is well-suited for real-time processing. The "Hello World" program in Motif follows (Johnson, E., 1994, p. 90):

```
/* Short Motif hello world program. */

#include <Xm/Xm.h>
#include <Xm/PushButton.h> /* XmPushButton */

void exitCB(Widget widget, XtPointer client_data, XtPointer call_data)

{ /* exitCB */

    exit(0);

} /* exitCB */

int main(int argc, char** argv)

{ /* main */
    Widget      parent;
    XtAppContext app_context;
    Widget      hello;

    parent = XtVaAppInitialize(
        &app_context, "UNIXReview", (XrmOptionDescList) NULL, 0, &argc, &argv
        (String *) NULL, NULL);

    hello = XtVaCreateManagedWidget(
        "hello", xmPushButtonWidgetClass, parent, NULL);

    XtAddCallback(hello, XmNactivateCallback, (XtCallbackProc) exitCB,
        (XtPointer) NULL);

    XtRealizeWidget(parent);
    XtAppMainLoop(app_context);

} /* main */
```

The following resource file works with the hello.c program:

```
! X resource file for UNIXReview apps
!
*hello.labelstring: Hello world from UNIX REVIEW
*hello foreground:  black
*hello background:  orange
*title:             UNIX REVIEW Tk Test
```

B. TCL/TK

Tcl is a shell interpreter which implements frequently used functions written in C. The interpreter can process the commands individually or as a script from a text file. The commands include the ability to access the standard types of simple and complex variables as well as control structures such as while and for loops and if and case statements. A unique aspect of *Tcl* is that the user can implement a frequently used function in C and build it into the *Tcl* interpreter. The new command can then be used in programming *Tcl* scripts or from the *Tcl* command line. (Ousterhout, 1994, pp. 7-23)

Tk, like *Motif*, is a library of C programs which implement widgets consistent with the Xwindows environment. The library is compiled with the *Tcl* interpreter, which adds a graphical programming capability to the interpreter. The user can easily build windows interfaces in scripts with very few lines of easily understood code, a significant advantage over *Motif*. (Ousterhout, 1994, pp. 145-155)

Because *Tcl/Tk* is both a high-level and interpreted language, resulting applications do not run as fast as compiled C programs. However, this is offset by rapid development time, ease of learning to program, and the ability to enter commands interactively. Since the language is not compiled, testing, debugging, and modification can be completed with immediate feedback, as opposed to significant delays while recompiling an application after each minor change. The "Hello World" program in *Tcl/Tk* follows (Johnson, E., 1994, p. 89), implementing the functionality of 20 lines of *Motif* code in six lines:

```
#!/local/bin/wish -f
# Tcl/Tk  hello world script.
#
button .button -text "Hello world from UNIX REVIEW"
.button configure -command "destroy ."
.button configure -foreground black -background orange
.button configure -activebackground orange2
wm title . "UNIX REVIEW Tk Test"
pack .button
```

C. BEST ALTERNATIVE

The most striking aspect of the comparison between *Motif/C* and *Tcl/Tk* is the reduction in code complexity when using the latter. The details of X are hidden from the

programmer by Tcl, but this could also limit flexibility in rare cases (the Tk library of widgets is very robust, so most requirements will be met without having to resort to creating new widgets or widget options). Since Tcl/Tk is a library of C programs, it can be extended to access the DIS network library functions in a similar fashion to a program developed with Motif. The ease of programming and modification, combined with access to a standard Xwindows interface, and the ability to use high-level commands interactively, favor the Tcl/Tk programming environment. However, the speed of the resulting application must be considered since the desired toolkit must process network traffic in real-time. Because of the processing power inherent in currently available Unix workstations, it was hypothesized that the speed issue was not significant when compared to the advantages of Tcl/Tk. Therefore Tcl/Tk was chosen as the software development environment for this project. Tcl/Tk has the added advantage of no monetary cost, since it is available at no charge from the University of California at Berkeley and several other Internet software archive sites. Tcl/Tk is distributed as source code which can be compiled on many types of Unix workstations. (Ousterhout, 1994, pp. 1-2)

V. IMPLEMENTATION AND EVALUATION

The DIS PDU Monitor was designed to provide a real-time graphical display of PDU traffic volumes according to PDU type. Taking advantage of the flexibility inherent in the Tcl programming environment, an extended Tk shell was built which can interact with a DIS network and a workstation's system clock. A Tk script was written to display graphical traffic volume charts based on data collected using the extended functions.

The resulting Xwindows-based tool was used to measure actual and simulated network loads to determine the performance of the PDU Monitor. The tests also served to measure the viability of the Tcl/Tk programming environment for real-time applications. The following sections described the design of the toolkit and the results of its testing.

A. TOOLKIT DESIGN

This section discusses the extension of Tk to include functions to access DIS networks and the use of the extended Tk shell to build a graphical application.

1. Tk Extensions

Tcl/Tk is a graphical shell built by linking together C functions and programs which provide commonly used capabilities into an interpreter which uses these routines as individual high-level commands. The commands can be used interactively on a command line or batch processed as a program script from a text file.

A C extension to Tk, written by AT&T Bell Labs, was available to draw various types of graph widgets. Functions to interact with the system clock were obtained from the TclX package written by Karl Lehenbauer and Mark Diekhans (available via anonymous ftp from harbor.ecn.purdue.edu:/pub/tcl). To interact with the network, C functions were written to access the DIS network code used by all NPSNET-IV applications. Tk was then modified to include the graph, clock, and network capabilities.

To extend the capabilities of Tk, functions to implement new commands must be written in C. The new functions must be declared in the *TkAppInit.c* program, which is

part of the Tk distribution, or in a separate *AppInit.c* program. The latter approach allows the distribution of the extension as a package which can be applied to different versions or extensions of Tk (Ousterhout, 1994, pp. 305-310). To implement this approach, the following two files were used:

a. netAppInit.c

This file is a modified version of *TkAppInit.c*, a generic C program to incorporate new commands into the Tk shell. Each command type is declared in the program's declarations. In the main program body each command is named and defined. This file is compiled and linked with *disnetlib.c*, the graph and clock functions, and the rest of the Tk functions to build a new interpreter called *netwish*. Source code for *netAppInit.c* is in Appendix A.

b. disnetlib.c

This file contains C functions to implement the DIS network access commands. Each function returns its result to the Tk interpreter. Source code for *disnetlib.c* is in Appendix B. The following functions are defined:

- **netopen** – initializes the program to access a DIS multicast network with optionally specified port, network group, time-to-live (TTL), and exercise identification parameters.
- **netopenbroadcast** – initializes the program to access a DIS broadcast network with an optionally specified exercise identification parameter.
- **pduread** – reads the network PDU queue to obtain the next PDU, and returns the PDU type and an unsigned integer representing the 32-bit network address of the host which sent the PDU.
- **pduwrite** – writes a PDU of the type specified to the network queue; not all of the 27 DIS PDU types are implemented since this command was added for demonstration and test purposes, but can be extended.
- **gethostid** – queries the network name server with a host name and obtains the 32-bit IP address for that host. (Stevens, 1990, pp. 264-265, 393-395)
- **netclose** – terminates network access.

2. PDU Monitor Design

The monitoring tool was built using the Tcl Stripchart function library written by BBN Systems and Technologies. These functions use the graph and clock functions to demonstrate an x-y graph which scrolls along the x-axis as time passes. It was necessary

to modify these routines to incorporate the DIS network commands to collect data to display on the graph, as well as to allow implementation of the features listed below.

The program provides the user with a button-controlled dialog box to choose the type of DIS network and to enter the desired network parameters, and then opens the network with those parameters (Figure 2 shows the parameter input window for multicast networks).

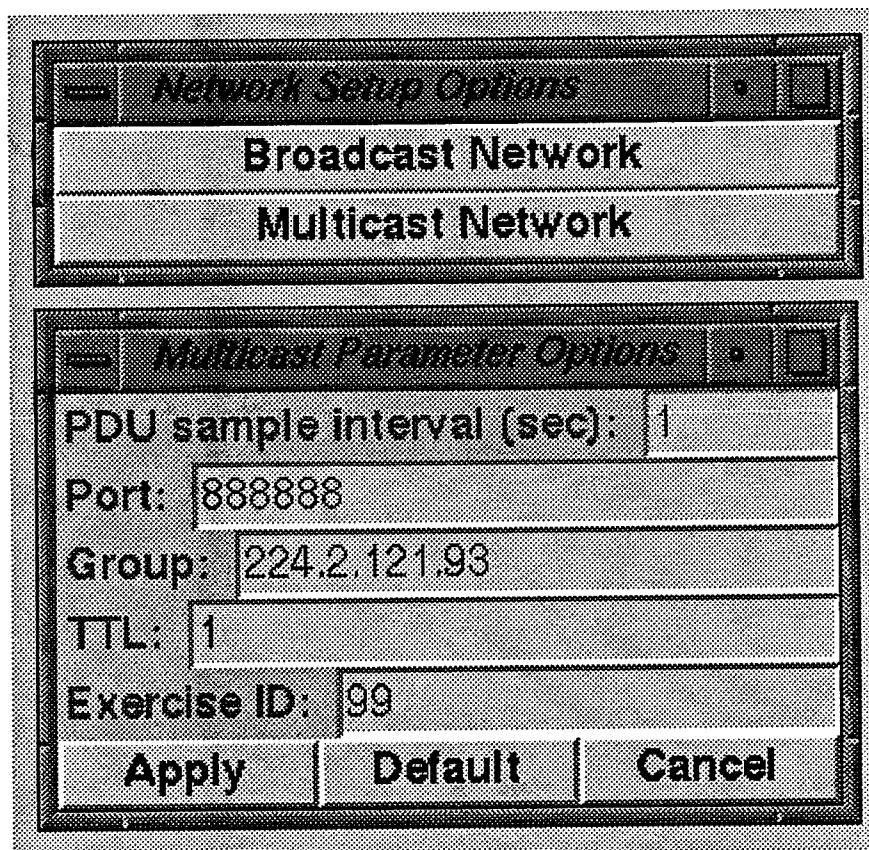


Figure 2. Multicast Network Parameter Input Window

The program then enters a "do forever" loop to collect PDUs from the network queue for a user-specified time interval, average the counts of each PDU type over that interval, update all visible graphs with the new averages, and check for user initiated menu selections. When the user selects the "Exit Program" menu option, the program performs a network close and exits. While it is running, the user may select any of the menu options to access the features described below or modify the display parameters for

the stripcharts. Appendix C contains the Tcl code to implement the PDU Monitor, and Appendix D contains instructions to obtain and install the program.

3. PDU Monitor Features

The PDU monitor is designed to be flexible enough to meet varied display requirements for the network traffic volumes. This flexibility is especially useful in localizing problems caused by specific hosts or traffic types when troubleshooting network irregularities, and is achieved through the following features:

- **Chart Prints:** The user can save any chart to a PostScript file at any time. The file is saved with a time stamp in its name and can be printed on the network printer.
- **Sampling Interval:** The user can vary the interval over which the network is sampled. Short sampling times provide feedback on every change in traffic volume, while longer intervals smooth the volume curve by showing an average rate.
- **Time Scaling:** Charts can be instantly modified to show the data over various lengths of time. Short time windows allow the user to see a detailed picture of the graph movements, or the charts may be configured to show longer periods of data collection with less detail.
- **Multiple Charts:** The user can display separate charts for different PDU types at the same time. This allows observation of data rates that are usually orders of magnitude apart, depending on the PDU type.
- **Host Selection:** The monitor can observe traffic from all hosts sending PDUs on the network, or the user can select an individual host to monitor.
- **Online Help:** A complete help system is available from the menu. Instructions are available for every menu option available, as well as general information about the program. A representative help window is shown in Figure 3.

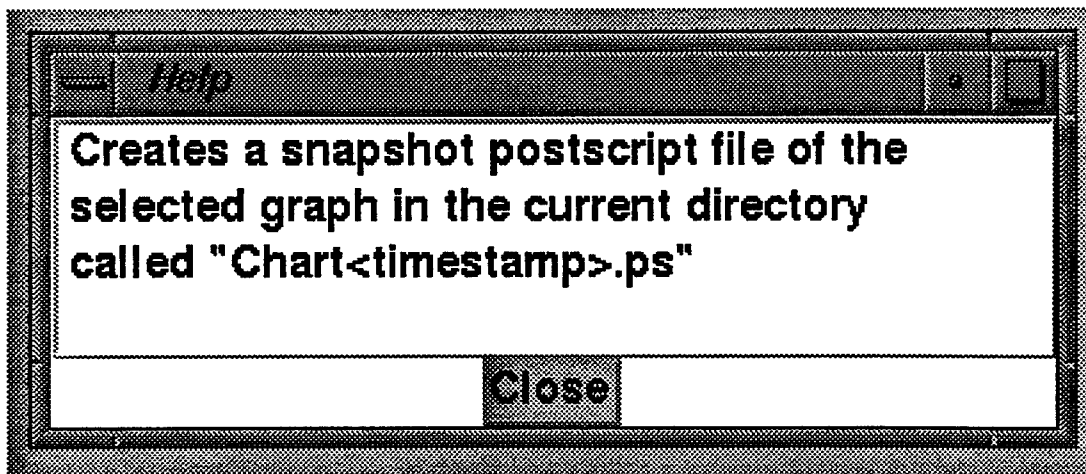


Figure 3. Help Window for "Print Chart" Menu Option

- **Xwindows Consistency:** The monitor's windows operate in the same manner as other window-oriented programs in the X environment. This allows the user to use all the window configuration options normally available, including stretching and shrinking of chart windows.

Figure 4 shows a PDU monitor chart – for the Entity State PDUs (one of the 27 DIS PDU types) – and several of the features listed above. The chart's x-axis label includes the size of the displayed window, in this case indicating that the last two minutes of data are visible. The y-axis shows the PDU rate after each sampling interval. For example, the peak load for Entity State PDUs (about 33 PDU/second) occurred at time 14:17:25. The graph title shows the last data rate value. Above the chart is the PDU Monitor Control Panel, which includes a menubar to access the various program features, as well as a status message section to report program activities. The status message in Figure 4 shows the hosts currently being monitored (all hosts) and the sampling interval chosen by the user (1 second). The standard X window control icons are visible across the top of each window, allowing stretching, moving, and changing the window to and from an icon.

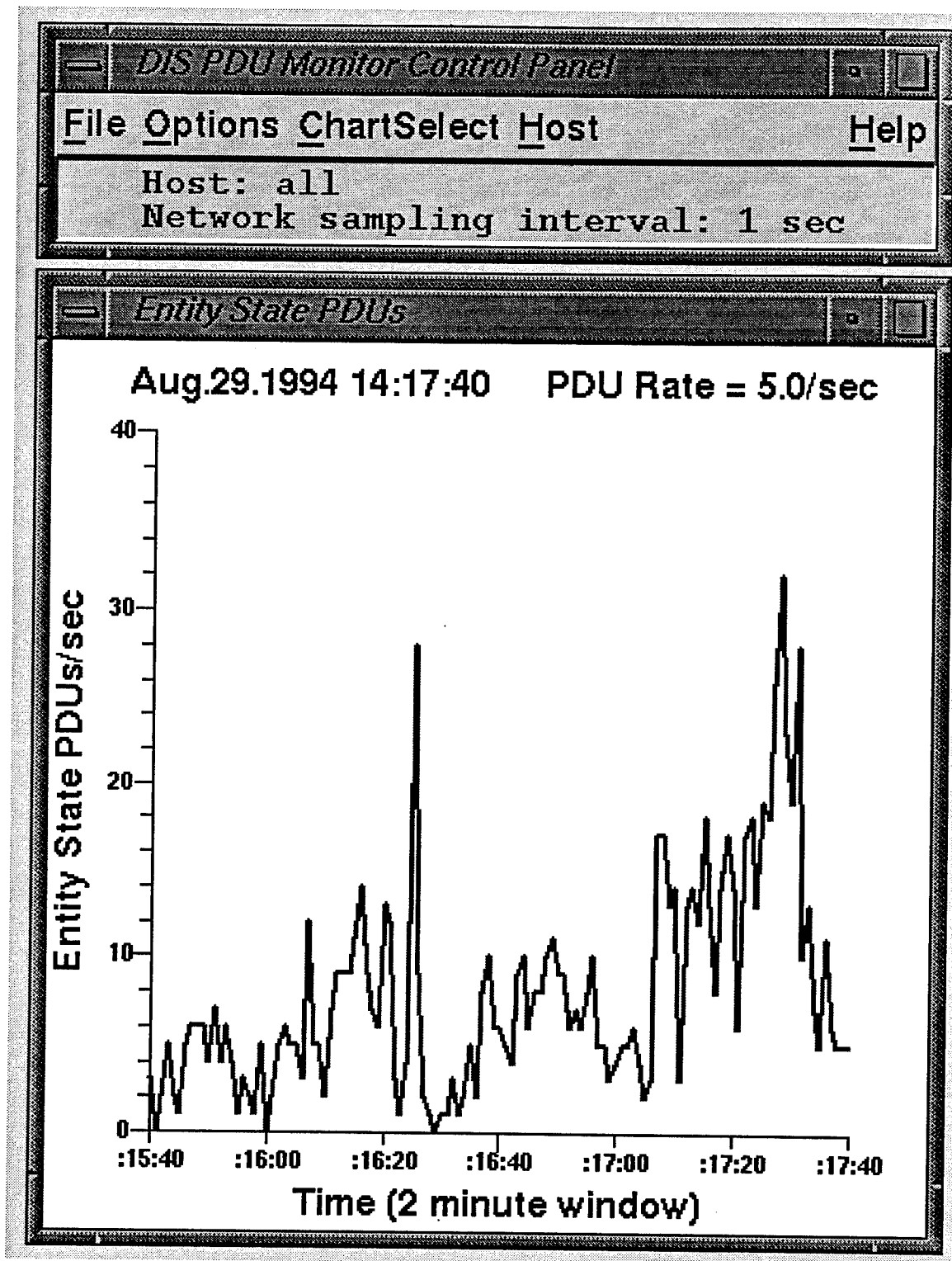


Figure 4. PDU Monitor Control Panel and PDU Chart

VI. TEST AND EVALUATION

Three areas were addressed during testing of the PDU Monitor. Where the user has opportunity to modify the operating configuration of the program, checks were implemented to ensure the modifications did not cause program errors. The program was observed to determine its response to unexpected situations, whether caused by user, computer, or the network. Finally, performance of the monitor was measured to determine its data handling capacity.

A. INPUT ERROR DETECTION

The PDU monitor was designed to take advantage of buttons and menu choices wherever possible to limit the possibility of incorrectly typed information. For text information that must be entered by the user, error detection and/or correction was built into the program. Input values are checked before the input window (Figure 5) is closed and the input accepted.

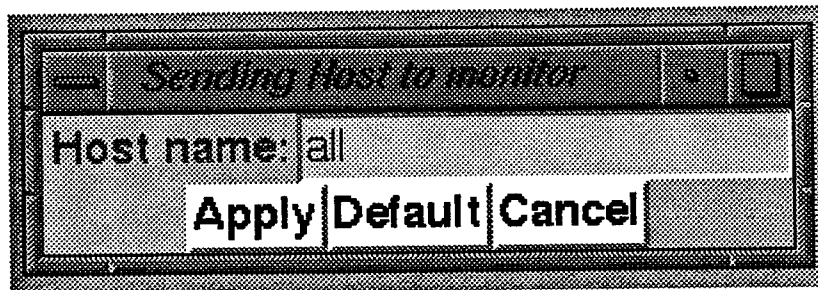


Figure 5. Text Input Window

Numbers such as the sampling interval, chart time window, TTL, and exercise identification are checked to preclude any extraneous non-numeric characters and to ensure the numbers are within a valid range. A popup error window notifies the user of the specific error and an opportunity for re-entry is provided. Non-integer values are simply rounded.

When a host name is entered to select a specific host to monitor, the name is passed to the network domain name server. If the server cannot locate that host on the network, it notifies the monitor program. However, it cannot be determined from the error whether the host does not exist (i.e., the name was spelled incorrectly), or if there is a problem with network connectivity from the computer the PDU monitor is running on. The user is advised to check the host name spelling, and if necessary, use a different computer with connectivity known to be good. This can be verified by using a Unix telnet, ftp, or ping command to the desired host.

There are some parameters, such as port number, multicast group, or exercise number, which may be valid in terms of entry requirements but not for the situation the user desires to monitor. For example, the user may enter exercise number 23 in the opening parameter selection window. A proper net open will be effected, but if the exercise is actually running with an identification of 25, the monitor will not see any traffic from that simulation. To correct such situations, the monitor includes a menu option to display all configurable parameters that are currently active. This window, shown in Figure 6, can be used to aid in troubleshooting unexpected monitor performance, such as when an exercise is underway but no traffic is reflected on the charts.

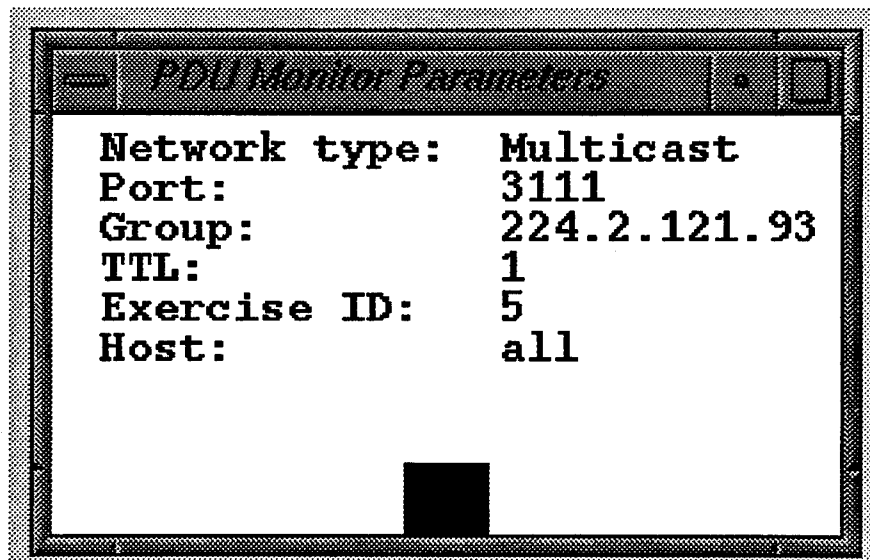


Figure 6. Configuration Parameter Display Window

B. EXCEPTION HANDLING

Most errors will occur from invalid user input. As noted above, the user will be notified of the error, shown the proper format for the input, and given the opportunity to correct the entry. Should he abort the entry process by pressing the cancel button, the program will proceed with the previous value for that parameter.

If the user attempts to print a chart that does not exist, or initiate a chart that already exists, the operation is not executed and he is notified of the error. Other existing charts will be unaffected.

Failure to open the network with the parameters entered by the user will cause the network code to return a failure message to the shell from which the monitor was started and will terminate the monitor program. In some cases, particularly when attempting to use a network port already being used by another program, the network code will not return control to the monitor program. In this case, the user must terminate the monitor with a break (control-c) and verify that all *netwish* processes in the originating shell are terminated (using the Unix "ps" command and killing any remaining *netwish* processes).

C. PDU MONITOR PERFORMANCE

The monitor has been tested using both live network exercises and simulated traffic loads. The use of actual exercises has demonstrated the validity of the data collection interface and its ability to monitor network traffic. However, since traffic loads in a real simulation are the result of a non-controlled environment, they could not be used to verify the correctness of the charts plotted by the PDU monitor. Additionally, the traffic loads generated during actual exercises were not large enough to approach the maximum capacity of the PDU monitor.

To obtain more concrete verification of the monitor charts' validity, a small Tcl script using the "pduwrite" command was written to generate specific traffic volumes, which were correctly reflected on the monitor charts. This script was also used to generate very large traffic volumes to attempt to determine the capacity of the monitor. Proper performance was verified at over 500 PDUs per second, but specific numbers were

impossible to obtain since the sending hosts overloaded their network queues when sending at rates between 200 and 300 per second. This points to a conclusion that the PDU monitor is more than capable of keeping up with any conceivable traffic load that can be generated by the current network configuration.

Figure 7 shows the results of test traffic loads applied to the PDU Monitor. The PDU rate was incrementally increased from 100 PDUs/second to 200, then 300, and finally

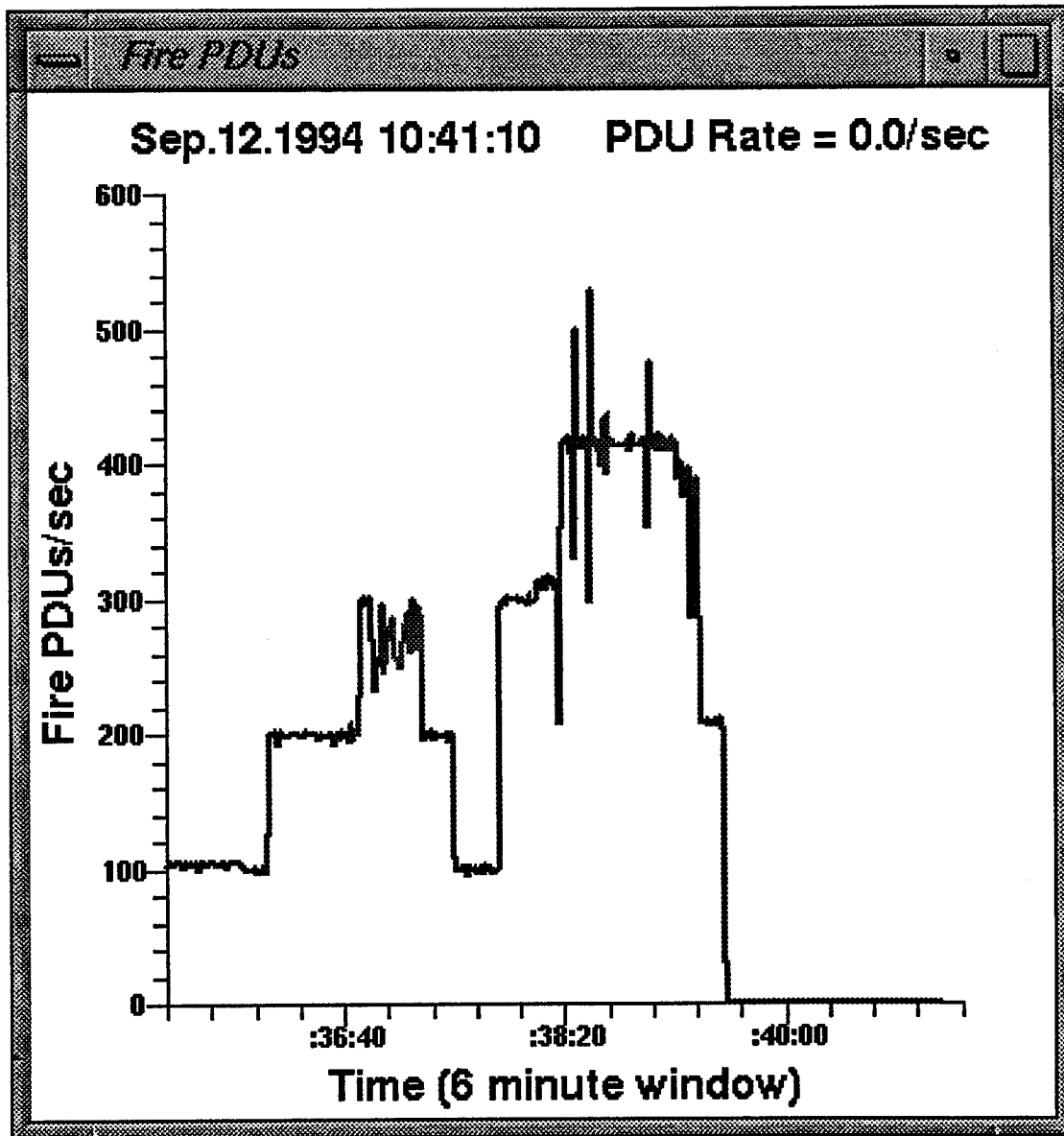


Figure 7. PDU Chart with Test Traffic Loads

400 per second. At the 400 rate, some aberrations can be seen as the sending hosts had difficulty sending PDUs at constant, high rates, as discussed above. A very high traffic volume was obtained, but could only be maintained for a short time, as shown in Figure 8. However, the Monitor did demonstrate its ability to process well over 600 PDUs per second, and the sharp drop reflected on both charts after traffic flow was terminated shows that no processing backlog was built up.

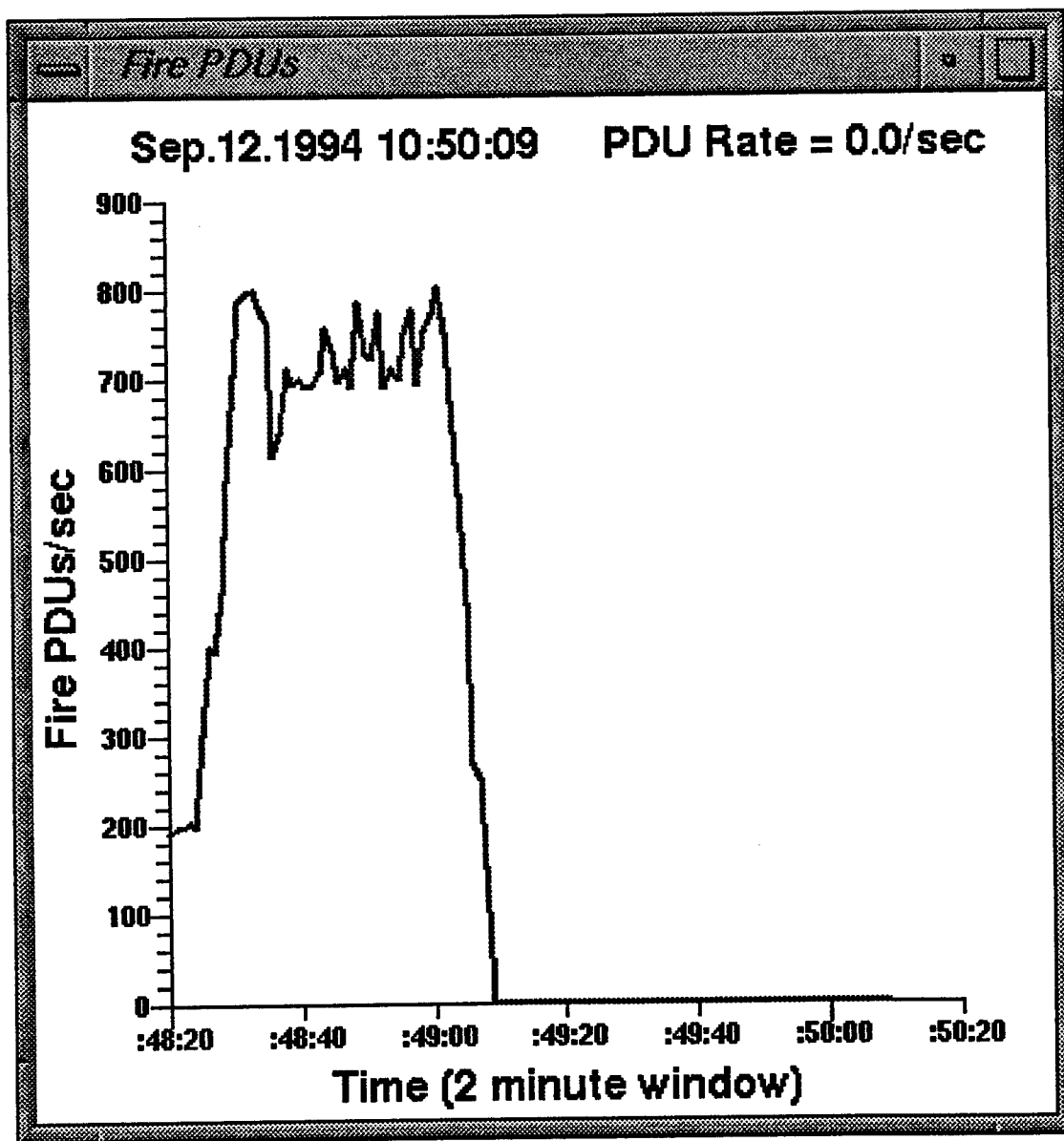


Figure 8. PDU Chart with Maximum Traffic Load

Performance of the monitor can be affected by several factors, though almost all performance loss will only be of concern when traffic volumes are abnormally high. Most degradation that can occur results from the computer performing other activities in addition to the PDU monitor. To obtain the best performance, the monitor should not be run on a computer that functions as a server for many user accounts or system activities. During a major exercise, routine use of the monitor's computer should be limited (as would the use of the machines actually running the simulation).

Keeping more than four or five charts open simultaneously will slow the monitor down at very high data rates (more than 200-300 per second); this will not usually be a problem since even large computer displays cannot adequately show large numbers of charts. Opening and closing charts will slightly affect the PDU counts as the opening or closing process is occurring (usually about a second), so the rate value will be lower than actual while the process occurred and slightly higher immediately after; this will only be noticeable if the network sampling rate is very small (1-3 seconds).

VII. FUTURE RESEARCH

Research for this thesis highlighted several areas for follow-on work related to developments in the use of virtual reality and networking. In addition to providing the *netwish* toolkit to access DIS networks, the efforts to build the toolkit demonstrated the utility of Tcl/Tk for expanding the capabilities of the toolkit. The PDU Monitor application that was developed also lends itself to continued use and expansion.

A. NETWISH

The fact that Tcl/Tk is an interpreted language allows the commands implemented for use in the PDU Monitor to be used interactively in the *netwish* shell or in applications built with scripts. The inherent flexibility of Tcl/Tk will allow the commands in *netwish* to be modified or new commands to be added.

1. Using netwish

The advantages of the Tcl/Tk shell have already been seen in this network application. To test the PDU Monitor, a simple script was written to generate false PDUs on a network. The *pduwrite* and *pduread* commands were also used interactively for testing purposes. Users desiring to interact with a DIS network, but not needing or wanting to run an entity simulator, can use *netwish* commands interactively or in a script to obtain precise results. This method provides complete control of the use of the network, whereas a simulator's sending and processing of traffic is variable and usually unknown.

2. Extending netwish

For this thesis, Tk was initially extended only to add the functionality necessary to monitor the network, since the PDU Monitor is a read-only application. When the limitations of existing methods of generating network traffic became obvious while testing the Monitor, the *pduwrite* function was easily added to Tk as well. This new function does not include all PDU types, but they can be added. The function puts

dummy data in the PDUs, but it could be modified to generate traffic that will actually affect a live simulation.

Implementation of new commands is only limited by the imagination. For example, the *gethostid* command was written in response to a desire to be able to selectively monitor hosts sending PDU traffic. As user requirements for the DIS toolkit change, it is easily adapted.

B. USING THE PDU MONITOR

The PDU Monitor provides the ability to determine the impact of various activities as they occur in the network environment. This will not only enable determination of current network capabilities, but will allow better preparation for future growth and development of software that makes better use of scarce assets.

The PDU Monitor has already been used by one developer to test, troubleshoot, and optimize the network access of his application. Developers should begin using this tool as soon as their program can send traffic on the network, so the resulting software will not unleash an unknown, and possibly unnecessary, traffic load.

The development of the PDU Monitor provides the tools for another researcher to make an exhaustive study of network activities, during both live and scripted simulation exercises. Particular attention should be paid to generating unusual traffic types or loads and monitoring the impact on the network.

C. MODIFYING THE PDU MONITOR

Since the PDU Monitor is implemented by a script contained in a simple text file, it can be easily modified to adapt to different display or configuration requirements. If new *netwish* commands are added, they can be quickly integrated into the Monitor.

The PDU Monitor can segregate traffic by PDU type or sending host. There are several interesting options for growth in this area. The *pduread* command returns the PDU type and sending host of each PDU. It does not read the bulk of the PDU structure, where information about the entity or message is stored. If the command were modified

to read other PDU fields, the Monitor could be made to segregate traffic based on such things as entity type, national flag, or location or movement in the virtual world.

Only one sending host can be monitored at a time, or the aggregate of all hosts. The checks for the hostname could be modified to check only as far as the domain name, so localized groups of hosts could be monitored. This might require some modification of the *gethostid* function.

As with the *netwish* shell itself, there is no limit to the number of ways this application can be changed. What is unique is that this application is built with an extensible toolkit and can be rapidly modified by users unfamiliar with the Xwindows implementation and programming details of their system.

VIII. CONCLUSIONS

The merging of virtual world and wargaming simulations into a coherent, distributed network environment allows the potential to drastically alter current paradigms in military training and weapons development. The effects will be most significant in areas that previously have involved potentially lethal environments, especially as the realism of simulations increases.

The joint operation of many entity and wargame simulations, linked in a single exercise via computer networks, can provide the necessary realism to achieve significant improvements in training and testing procedures. However, the resulting changes in the network activity which will accompany the advancements in these simulations will also be drastic and to date unknown, at least on a real-time basis. The development of the PDU Monitor and tools for other DIS network applications will allow measurement of the current network environment. The tools are also now available for software developers to evaluate the impact that their programs will have on the network as they build applications to meet current and future requirements.

The goal of this thesis has been to develop tools which provide detailed real-time knowledge of the impact of distributed simulations on networks. Such knowledge is critical as a basis for intelligent decision making regarding development of the DIS network architecture.

APPENDIX A. C CODE TO ADD DIS NETWORK FUNCTIONS TO TK

```
/* File:          netAppInit.c
 * Description:    Provides a version of the Tcl_AppInit procedure to
 *                extend Tk to include DIS network functions
 * Revision:      1.0 - 12Aug94
 *
 * Author:        Mitch Turner
 *                Code 3A, Naval Postgraduate School
 *
 * Internet:      mktturner@nps.navy.mil
 *
 * Copyright (c) 1993 The Regents of the University of California.
 * All rights reserved.
 *
 * Permission is hereby granted, without written agreement and without
 * license or royalty fees, to use, copy, modify, and distribute this
 * software and its documentation for any purpose, provided that the
 * above copyright notice and the following two paragraphs appear in
 * all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
 * FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
 * ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN
 * IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
 * CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT,
 * UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
 */

#ifndef lint
static char rcsid[] = "$Header: /user6/ouster/wish/RCS/tkAppInit.c,v 1.8
93/08/26 14:38:24 ouster Exp $ SPRITE (Berkeley)";
#endif /* not lint */

#include "tk.h"

/*
 * The following variable is a special hack that allows applications
 * to be linked using the procedure "main" from the Tk library.  The
 * variable generates a reference to "main", which causes main to
 * be brought in from the library (and all of Tk and Tcl with it).
 */
```

```

extern int main();
int *tclDummyMainPtr = (int *) main;

/* Declarations for xygraph widget.  Requires file graph.c implementing
 *   Bell Labs graph widget.
 */

extern Tcl_CmdProc GraphCmd;

/* Declarations for clock functions.  Requires following files from tclX
7.3:
 *   tclExtdInt.h      tclExtend.h      tclXconfig.h      tclXclock.c
 *   tclXcnvclock.c   tclXgetdate.c   tclXutil.c
 */

extern Tcl_CmdProc Tcl_GetclockCmd;
extern Tcl_CmdProc Tcl_FmtclockCmd;
extern Tcl_CmdProc Tcl_ConvertclockCmd;

/* Declarations for DIS net functions.  Requires disnetlib.c
 */

extern Tcl_CmdProc dis_NetOpenCmd;
extern Tcl_CmdProc dis_NetOpenBcastCmd;
extern Tcl_CmdProc dis_PduReadCmd;
extern Tcl_CmdProc dis_NetCloseCmd;
extern Tcl_CmdProc dis_PduWriteCmd;
extern Tcl_CmdProc dis_GetHostCmd;

/*
 *-----
 *
 *   Tcl_AppInit --
 *
 *   This procedure performs application-specific initialization.
 *   Most applications, especially those that incorporate additional
 *   packages, will have their own version of this procedure.
 *
 * Results:
 *   Returns a standard Tcl completion code, and leaves an error
 *   message in interp->result if an error occurs.
 *
 * Side effects:
 *   Depends on the startup script.
 *-----
 */

int
Tcl_AppInit(interp)
    Tcl_Interp *interp;      /* Interpreter for application. */
{
    Tk_Window main;
    main = Tk_MainWindow(interp);

```

```

/*
 * Call the init procedures for included packages. Each call should
 * look like this:
 *
 * if (Mod_Init(interp) == TCL_ERROR) {
 *     return TCL_ERROR;
 * }
 *     where "Mod" is the name of the module.
 */

if (Tcl_Init(interp) == TCL_ERROR) {return TCL_ERROR;}
if (Tk_Init(interp) == TCL_ERROR) {return TCL_ERROR;}
/*
 * Call Tcl_CreateCommand for application-specific commands, if
 * they weren't already created by the init procedures called above.
 */

/* Add xygraph command. Note ClientData argument of "main" vice NULL. */
Tcl_CreateCommand(interp, "xygraph", GraphCmd, (ClientData) main,
    (Tcl_CmdDeleteProc *) NULL);

/* Add clock commands */
Tcl_CreateCommand(interp, "getclock", Tcl_GetclockCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "fmtclock", Tcl_FmtclockCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "convertclock", Tcl_ConvertclockCmd,
    (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);

/* Add DIS network commands */
Tcl_CreateCommand(interp, "netopen", dis_NetOpenCmd, (ClientData) NULL,
    (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "netopenbcast", dis_NetOpenBcastCmd,
    (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "pduread", dis_PduReadCmd, (ClientData) NULL,
    (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "netclose", dis_NetCloseCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "pduwrite", dis_PduWriteCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "gethostid", dis_GetHostCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);

/*
 * Specify a user-specific startup file to invoke if the application
 * is run interactively. Typically the startup file is "~/.apprc"
 * where "app" is the name of the application. If this line is
 * deleted then no user-specific startup file will be run under any
 * conditions.
 */
tcl_RcFileName = "~/.netwishrc";
return TCL_OK;
}
/* End netAppInit.c */

```

APPENDIX B. C FUNCTIONS TO IMPLEMENT DIS NETWORK ACCESS

```

/* File:          disnetlib.c
 * Description:    Functions to give Tk DIS access capabilities.
 * Revision:      1.0 - 12Aug94
 *
 * Reference:     Military Standard--Protocol Data Units for Entity
 *               Information and Entity Interaction in a Distributed
 *               Interactive Simulation (DIS) (30Oct91)
 *
 *               IEEE P1278 (DIS 2.0)
 *
 *               Tcl and the Tk Toolkit - Ousterhout
 *
 * Author:        Mitch Turner
 *               Code 3A, Naval Postgraduate School
 *
 * Internet:      mktturner@nps.navy.mil
 */

#include "tcl.h"
#include "disdefs.h"
#include <netdb.h>

/*****
/* NetOpenCmd - netopen()
*****/
/* Arguments : multicast addressing info
 *
 * Description : This function opens a DIS multicast network,
 *               establishing the connection for sending and receiving.
 */
/*****
int
dis_NetOpenCmd (clientData, interp, argc, argv)
    ClientData clientData;
    Tcl_Interp *interp;
    int         argc;
    char        **argv;
{
    /* Command line option */

    int op = 0;
    extern char *optarg;

    /* Multicast defaults */

    char * port;

```


APPENDIX B. C FUNCTIONS TO IMPLEMENT DIS NETWORK ACCESS

```

/* File:          disnetlib.c
 * Description:    Functions to give Tk DIS access capabilities.
 * Revision:      1.0 - 12Aug94
 *
 * Reference:     Military Standard--Protocol Data Units for Entity
 *               Information and Entity Interaction in a Distributed
 *               Interactive Simulation (DIS) (30Oct91)
 *
 *               IEEE P1278 (DIS 2.0)
 *
 *               Tcl and the Tk Toolkit - Ousterhout
 *
 * Author:        Mitch Turner
 *               Code 3A, Naval Postgraduate School
 *
 * Internet:      mkturner@nps.navy.mil
 */

#include "tcl.h"
#include "disdefs.h"
#include <netdb.h>

/*****
 * NetOpenCmd - netopen()
 *****/
/* Arguments : multicast addressing info
 *
 * Description : This function opens a DIS multicast network,
 *               establishing the connection for sending and receiving.
 */
/*****
int
dis_NetOpenCmd (clientData, interp, argc, argv)
    ClientData clientData;
    Tcl_Interp *interp;
    int      argc;
    char     **argv;
{
    /* Command line option */

    int op = 0;
    extern char *optarg;

    /* Multicast defaults */

    char * port;

```

```

/*
 * Call the init procedures for included packages. Each call should
 * look like this:
 *
 * if (Mod_Init(interp) == TCL_ERROR) {
 *     return TCL_ERROR;
 * }
 *     where "Mod" is the name of the module.
 */

if (Tcl_Init(interp) == TCL_ERROR) {return TCL_ERROR;}
if (Tk_Init(interp) == TCL_ERROR) {return TCL_ERROR;}
/*
 * Call Tcl_CreateCommand for application-specific commands, if
 * they weren't already created by the init procedures called above.
 */

/* Add xygraph command. Note ClientData argument of "main" vice NULL. */
Tcl_CreateCommand(interp, "xygraph", GraphCmd, (ClientData) main,
    (Tcl_CmdDeleteProc *) NULL);

/* Add clock commands */
Tcl_CreateCommand(interp, "getclock", Tcl_GetclockCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "fmtclock", Tcl_FmtclockCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "convertclock", Tcl_ConvertclockCmd,
    (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);

/* Add DIS network commands */
Tcl_CreateCommand(interp, "netopen", dis_NetOpenCmd, (ClientData) NULL,
    (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "netopenbcast", dis_NetOpenBcastCmd,
    (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "pduread", dis_PduReadCmd, (ClientData) NULL,
    (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "netclose", dis_NetCloseCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "pduwrite", dis_PduWriteCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateCommand(interp, "gethostid", dis_GetHostCmd, (ClientData)
    NULL, (Tcl_CmdDeleteProc *) NULL);

/*
 * Specify a user-specific startup file to invoke if the application
 * is run interactively. Typically the startup file is "~/.apprc"
 * where "app" is the name of the application. If this line is
 * deleted then no user-specific startup file will be run under any
 * conditions.
 */
tcl_RcFileName = "~/.netwishrc";
return TCL_OK;
}
/* End netAppInit.c */

```

```

/* including the one returned to */
/* pdu, if any; 0 if no pdu's */

if (nodes == -1) {
    printf("pduread(): Error on net_read()\n");
} else if (nodes != 0) {
    sprintf(interp->result, "%d %u", type, addr.s_addr);
    freePDU(pdu);
} else {
    sprintf(interp->result, "x");
}

return TCL_OK;
}
/*****
/* GetHostCmd - gethostid()
*****/
/* Arguments : hostname
*
* Description : This function queries the network name server for the
*               integer IP address of "hostname"
*/
*****/
int
dis_GetHostCmd (clientData, interp, argc, argv)
    ClientData clientData;
    Tcl_Interp *interp;
    int argc;
    char **argv;
{
    register char *ptr;
    register struct hostent *hostptr;
    char **listptr;
    struct in_addr *addr_ptr;

    if (argc == 2) {
        ptr = argv[1];
        if ((hostptr = gethostbyname(ptr)) == NULL) {
            printf("gethostid(): Error on gethostbyname()\n");
            sprintf(interp->result, "error");
        } else {
            listptr = hostptr->h_addr_list;
            addr_ptr = (struct in_addr *) *listptr;
            sprintf(interp->result, "%u", *addr_ptr);
        }
    } else {
        printf("gethostid(): Wrong number of arguments\n");
    }
    return TCL_OK;
}
/*****
/* NetCloseCmd - netclose()
*****/
/* Arguments : none
*

```

```

    * Description : This function closes the active DIS network.
    */
    /*****
int
dis_NetCloseCmd (clientData, interp, argc, argv)
    ClientData  clientData;
    Tcl_Interp *interp;
    int         argc;
    char        **argv;
{
    net_close();
    return TCL_OK;
}
    *****/
    /* PduWriteCmd - pduwrite() */
    /*****
/* Arguments : pdu type to send
*
* Description : This function sends a PDU on the active DIS network.
*               Not all DIS PDU types are included, since this was
*               created for test and demonstration (but more can be
*               added).
*/
    *****/
int
dis_PduWriteCmd (clientData, interp, argc, argv)
    ClientData  clientData;
    Tcl_Interp *interp;
    int         argc;
    char        **argv;
{
    int          i = 0;
    unsigned short host_id = 11;
    char         *pdu;
    PDUType      type;

                                /* example PDUs */
    EntityStatePDU *ESpdu;
    FirePDU        *Fpdu;
    DetonationPDU  *Dpdu;
    ServiceRequestPDU *SRpdu;
    ResupplyPDU    *Rpdu;
    ResupplyCancelPDU *RCpdu;
    RepairCompletePDU *RC_pdu;
    RepairResponsePDU *RRpdu;
    CollisionPDU    *Cpdu;

    ArticulatParamsNode *APNptr;          /* list nodes */
    SupplyQtyNode *SQNptr;

    if (argc == 2) {
        type = atoi(argv[1]);
    } else {
        type = 1;
    }
}

```

```

switch (type) {

case (CollisionPDU_Type):
    Cpdu = (CollisionPDU *) mallocPDU(CollisionPDU_Type);
    pdu = (char *) Cpdu;
    break;

case (RepairResponsePDU_Type):
    RRpdu = (RepairResponsePDU *) mallocPDU(RepairResponsePDU_Type);
    pdu = (char *) RRpdu;
    break;

case (RepairCompletePDU_Type):
    RC_pdu = (RepairCompletePDU *) mallocPDU(RepairCompletePDU_Type);
    pdu = (char *) RC_pdu;
    break;

case (ResupplyCancelPDU_Type):
    RCpdu = (ResupplyCancelPDU *) mallocPDU(ResupplyCancelPDU_Type);
    pdu = (char *) RCpdu;
    break;

case (ResupplyOfferPDU_Type):          /* same structure employed */
case (ResupplyReceivedPDU_Type):      /* by both */
    Rpdu = (ResupplyPDU *) mallocPDU(ResupplyOfferPDU_Type);
    if (SQNptr = attachSupplyQtyNode((char *) Rpdu,
                                     ResupplyOfferPDU_Type)) {
        SQNptr->supply_quantity.quantity = 4.8;
    }
    pdu = (char *) Rpdu;
    break;

case (ServiceRequestPDU_Type):
    SRpdu = (ServiceRequestPDU *) mallocPDU(ServiceRequestPDU_Type);
    if (SQNptr = attachSupplyQtyNode((char *) SRpdu,
                                     ServiceRequestPDU_Type)) {
        SQNptr->supply_quantity.quantity = 7.2;
    }
    pdu = (char *) SRpdu;
    break;

case (DetonationPDU_Type):
    Dpdu = (DetonationPDU *) mallocPDU(DetonationPDU_Type);
    if (APNptr = attachArticulatParamsNode((char *) Dpdu,
                                           DetonationPDU_Type)) {
        /* Success, fill in some data */
        APNptr->articulat_params.change = (unsigned short) 11;
        APNptr->articulat_params.parameter_value[7] = 0xEC;
    }

    pdu = (char *) Dpdu;
    break;

case (FirePDU_Type):
    Fpdu = (FirePDU *) mallocPDU(FirePDU_Type);

```

```

/* first field after header */
Fpdu->firing_entity_id.address.site = (unsigned short) 17;
/* last field */
Fpdu->range = 3.0;

pdu = (char *) Fpdu;
break;

case (EntityStatePDU_Type):
    ES pdu = (EntityStatePDU *) mallocPDU(EntityStatePDU_Type);

    /* first field after header */
    ES pdu->entity_id.address.site = SITE_ID_NPS;
    ES pdu->entity_id.address.host = host_id;

    ES pdu->force_id = ForceID_White;
    ES pdu->alt_entity_type.extra = (Extra) 0xDD;
    ES pdu->entity_orientation.psi = (float) 0xAA;
    ES pdu->entity_orientation.theta = (float) 0xBB;
    ES pdu->entity_orientation.phi = (float) 0xCC;
    ES pdu->dead_reckon_params.algorithm = DRAlgo_DRM_FVW;
    ES pdu->dead_reckon_params.linear_accel[0] = 1.1;
    ES pdu->dead_reckon_params.angular_velocity[0] = 2;

    strcpy(ES pdu->entity_marking.markings, "GO DIS!");

    /* Add some nodes */
    for (i = 0; i < 2; i++) {
        if (APNptr = attachArticulatParamsNode((char *) ES pdu,
                                                EntityStatePDU_Type)) {
            /* Success, fill in some data */
            APNptr->articulat_params.change = (unsigned short) i+6;
            APNptr->articulat_params.parameter_value[7] = 0xEE;
        }
    }

    pdu = (char *) ES pdu;
    break;

default:
    printf("default case reached!\n");

} /* end switch(type) */

if (net_write(pdu, type) == FALSE)
    sprintf(stderr, "net_write() failed\n");
freePDU(pdu);

return TCL_OK;
}
/**** End disnetlib.c *****/

```

APPENDIX C. TCL SCRIPTS IMPLEMENTING THE DIS PDU MONITOR

```
#####
#File:      dis.tcl
#Contents:  Routines to run DIS PDU monitor stripcharts
#System:
#Created:   10-Jul-1994
#Author:    Mitchell Turner
#
#Remarks:  Uses routines in BBN Systems and Technologies stripchart
#           library.
#           Requires tcl/tk, extended with tclX clock functions and DIS
#           interface functions.
#
#####

#####
# Procedure:  popup_error
# Description: Creates an error message in a popup window
# Requires:   nothing
# Arguments:  error text
# Returns:    nothing
# Sideeffects: A popup window is created
# Called by:  printchart, net_options, new_chart, fixed_window_proc,
#            win_size_hook, get_interval
#####
proc popup_error { text } {

    catch {destroy .errMsg}
    toplevel .errMsg
    wm title .errMsg "Error Message"
    wm geometry .errMsg +500+500

    frame .errMsg.top
    frame .errMsg.bottom

    label .errMsg.top.icon -bitmap error -background red

    message .errMsg.top.txt -font -*-Times-*-r-*-180-* \
        -justify center -text $text

    button .errMsg.bottom.b -text "Acknowledged" \
        -command {destroy .errMsg} -background gray90

    pack .errMsg.top.icon -side left -padx 20
    pack .errMsg.top.txt -side left

    pack .errMsg.bottom.b
```

```

pack append .errMsg .errMsg.top {top}
pack append .errMsg .errMsg.bottom {top}

BEEP
}

proc BEEP { } {
    puts stdout "\007" nonewline
}

#####
# Procedure:    printchart
# Description:  saves chart in postscript file in current directory
# Requires:    nothing
# Arguments:    type number of chart to print
# Returns:     nothing
# Sideeffects:  postscript file is created
# Called by:   make_top_menubar (on user choice)
#####
proc printchart { type } {

    global type_names

    if {[winfo exists .$type.graphWidgetFrame.graph]} {

        .$type.graphWidgetFrame.graph postscript \
            ../charts/[fmtclock [getclock] "Chart.%b%d.%X"].ps

    } else {

        popup_error "[lindex $type_names [expr $type-1]] PDUs chart does
not exist"
    }
}

#####
# Procedure:    exit_program
# Description:  destroys main windows, closes network, and exits netwish
# Requires:    nothing
# Arguments:    none
# Returns:     does not return
# Sideeffects:  program will exit
# Called by:   user menu choice
#####
proc exit_program {} {

    global pdu_types

    destroy .menub
    destroy .statusMsg
    foreach type $pdu_types {
        destroy .$type
    }
    netclose
    exit
}

```



```

}

#####
# Procedure:    build_info_hook
# Description:  procedure to create a new stripchart for a monitor window
# Requires:    nothing
# Arguments:    list of types of chart to start
# Returns:     nothing
# Sideeffects:  creates new stripchart widgets
# Called by:   StartupGGM
#####
proc build_info_hook {type_list} {

    global now timeWindow type_names

    set now [getclock]

    set xMax [xMaxTick $now $timeWindow]
    set xMin [expr $xMax - $timeWindow]

    foreach type $type_list {

        xygraph $.type.graphWidgetFrame.graph \
            -title "[fmtclock [getclock] "%b %d %Y %H:%M:%S"]" \
            -xlabel "Time" -xformatcommand timeformat \
            -xmin $xMin -xmax $xMax \
            -ylabel "[lindex $type_names [expr $type-1]] PDUs/sec" \
            -ymin "" -ymax ""

        pack append $.type.graphWidgetFrame \
            $.type.graphWidgetFrame.graph {top fill expand}

    }

    update

}

#####
# Procedure:    show_params
# Description:  procedure to show network environment parameters
# Requires:    nothing
# Arguments:    none
# Returns:     nothing
# Sideeffects:  none
# Called by:   user menu choice
#####
proc show_params { } {

    global net port group ttl ex_id host

    switch $net {

        1 {set nettype Broadcast
            set params " Network type:  $nettype\n Exercise ID:   $ex_id\n\

```

```

        Host:          $host\n\n"
    }
    2 {set nettype Multicast
        set params " Network type: $nettype\n Port:          $port\n\
        Group:        $group\n TTL:          $ttl\n\
        Exercise ID:   $ex_id\n Host:        $host\n\n"
    }
}

catch {destroy .showParams}
toplevel .showParams
wm title .showParams "PDU Monitor Parameters"
wm geometry .showParams +130+250

message .showParams.msg -font *-Courier*-r*--*-120-* -width 40c \
    -justify left -text $params

button .showParams.ok -text "OK" -command {destroy .showParams} \
    -background blue

pack .showParams.msg .showParams.ok -side top
}

#####
# Procedure:    net_setup
# Description:  proc to set network environment parameters on startup
# Requires:    nothing
# Arguments:    none
# Returns:     nothing
# Sideeffects:  vars port, group, ttl, ex_id, interval are updated
# Called by:   main script
#####
proc net_setup { } {

global net

    catch {destroy .netSetup}
    toplevel .netSetup
    wm geometry .netSetup +500+400
    wm title .netSetup "Network Setup Options"

    button .netSetup.bcast -text "Broadcast Network" -width 32 \
        -command { set net 1; net_options 1 }
    button .netSetup.mcast -text "Multicast Network" -width 32 \
        -command { set net 2; net_options 2 }

    pack .netSetup.bcast .netSetup.mcast -side top

    tkwait window .netSetup
}

#####
# Procedure:    net_options

```

```

# Description: proc to get network environment parameters on startup
# Requires:    nothing
# Arguments:   flag for broadcast (1) or multicast (2) network
# Returns:     nothing
# Sideeffects: vars port, group, ttl, ex_id, interval are updated
# Called by:   net_setup
#####
proc net_options { netflag } {

    global port group ttl ex_id interval e1 e2 e3 e4 e5

    catch {destroy .netParams}
    toplevel .netParams
    if {$netflag == 1} {
        wm title .netParams "Broadcast Parameter Options"
    } else {
        wm title .netParams "Multicast Parameter Options"
    }
    wm geometry .netParams +500+500

# Setup frames, labels, and entries in vars for packing in window

    set f1 [frame .netParams.1]
    set f2 [frame .netParams.2]
    set f3 [frame .netParams.3]
    set f4 [frame .netParams.4]
    set f5 [frame .netParams.5]

    set l1 [label .netParams.l1 -background grey90 -foreground red \
        -text "Port: "
    ]
    set l2 [label .netParams.l2 -background grey90 -foreground red \
        -text "Group: "
    ]
    set l3 [label .netParams.l3 -background grey90 -foreground red \
        -text "TTL: "
    ]
    set l4 [label .netParams.l4 -background grey90 -foreground red \
        -text "Exercise ID: "
    ]
    set l5 [label .netParams.l5 -background grey90 -foreground red \
        -text "PDU sample interval (sec): "
    ]

    set e1 [entry .netParams.e1 -background gray95 -relief sunken -width
10 ]

    set e2 [entry .netParams.e2 -background gray95 -relief sunken -width
15 ]

    set e3 [entry .netParams.e3 -background gray95 -relief sunken -width
5 ]

    set e4 [entry .netParams.e4 -background gray95 -relief sunken -width
5 ]

```

```

set e5 [entry .netParams.e5 -background gray95 -relief sunken -width
5 ]

$e1 insert 0 $port
$e2 insert 0 $group
$e3 insert 0 $ttl
$e4 insert 0 $ex_id
$e5 insert 0 $interval

# Build button row for window

set fb [frame .netParams.bottom]
set a [button $fb.apply -text "Apply" -width 10 \
-command {
    global port group ttl ex_id interval e1 e2 e3 e4 e5
    set port [$e1 get]
    set group [$e2 get]
    set ttl [$e3 get]
    set ex_id [$e4 get]
    set interval [$e5 get]
    set errflag [catch {set interval [expr int($interval)]}]
    if {($interval < 1) || ($interval > 60) || ($errflag != 0)} {
        set interval 1
        popup_error "Network sampling interval must be 1 - 60
seconds"
    } else {
        destroy .netParams
        destroy .netSetup
    }
}]
set d [button $fb.def -text "Default" -width 10 \
-command {global e1 e2 e3 e4 e5
    $e1 delete 0 end;$e1 insert 0 "888888"
    $e2 delete 0 end;$e2 insert 0 "224.2.121.93"
    $e3 delete 0 end;$e3 insert 0 "1"
    $e4 delete 0 end;$e4 insert 0 "99"
    $e5 delete 0 end;$e5 insert 0 "1"
}]
set c [button $fb.cancel -text "Cancel" -width 10 \
-command {destroy .netParams}]

# Put window pieces together

pack append $fb $a {left expand} $d {left expand} $c {right expand}

pack append $f1 $l1 {left} $e1 {right expand fill}
pack append $f2 $l2 {left} $e2 {right expand fill}
pack append $f3 $l3 {left} $e3 {right expand fill}
pack append $f4 $l4 {left} $e4 {right expand fill}
pack append $f5 $l5 {left} $e5 {right expand fill}

pack append .netParams $f5 {top fill}
if {$netflag == 2} {

```

```

        pack append .netParams $f1 {top fill} $f2 {top fill} $f3 {top
fill}
    }
    pack append .netParams $f4 {top fill} $fb {bottom expand fill}

    tkwait window .netParams
}
##### End proc net_options

#####
# Procedure:    new_chart
# Description:  proc to build new stripchart widget for another PDU type
# Requires:    nothing
# Arguments:    PDU type to begin monitoring
# Returns:     nothing
# Sideeffects:  line "tagID" is updated
# Called by:   user menu choice
#####
proc new_chart {type} {

    global pdu_types type_names host interval statusMsg

    if {[wininfo exists .$type]} {

        ShowWindow $type

        StartupGGM $type

        assoc_up_hook $type

        lappend pdu_types $type

        set statusMsg "Host: $host\nNetwork sampling interval: $interval
sec"
    } else {
        popup_error "[lindex $type_names [expr $type-1]] PDUs chart already
exists"
    }
}

##### End proc new_chart

##### Main tcl script

wm withdraw .
wm geometry . +130+150

# load stripchart library and initialize variables

source "stripchart.tcl"

# *** Note: to add/change PDU types, change rate vars here and
#         in getXYVal_proc. Change count vars here (set to zero).
#         Modify File and Charts buttons in proc make_top_menubar.

```

```

#      Modify type_names list creation below.

# Following rate globals must be in getXYVal_proc also

global rate1 rate2 rate3 rate4 rate5 rate6 rate7 rate8 rate9 rate10
global rate11 rate12 rate13 rate14 rate15 rate16 rate17 rate18 rate19
global rate20 rate21 rate22 rate23 rate24 rate25 rate26 rate27

global port group ttl ex_id interval host hostid type_names pdu_types
global net statusMsg

# Type names must be in order of numerical precedence as in pdu.h

set type_names { "Entity State" "Fire" "Detonation" "Collision" }
lappend type_names "Service Request" "Resupply Offer" "Resupply
Received"
lappend type_names "Resupply Cancel" "Repair Complete" "Repair Response"
lappend type_names "Create Entity" "Remove Entity" "Start Resume"
lappend type_names "Stop Freeze" "Acknowledge" "Action Request"
lappend type_names "Action Response" "Data Query" "Set Data"
lappend type_names "Data" "Event Report" "Message"
lappend type_names "Emission" "Laser" "Transmitter" "Signal" "Receiver"

set pdu_types {1}
set interval 1

set port 888888
set group 224.2.121.93
set ttl 1
set ex_id 99
set host all
set hostid 0

set count1 0; set count2 0; set count3 0; set count4 0; set count5 0
set count6 0; set count7 0; set count8 0; set count9 0; set count10 0
set count11 0; set count12 0; set count13 0; set count14 0; set
count15 0
set count16 0; set count17 0; set count18 0; set count19 0; set
count20 0
set count21 0; set count22 0; set count23 0; set count24 0; set
count25 0
set count26 0; set count27 0

# Get network parameters

set net 1

net_setup

# Startup graphic view manager

StartupGGM $pdu_types

first_time_assoc_up_hook

```

```

# Establish network connection

switch $net {

    1 {netopenbcast -e $ex_id}
    2 {netopen -p $port -I $group -t $ttl -e $ex_id}
}

set statusMsg "Host: $host\nNetwork sampling interval: $interval sec"

# Do forever

while 1 {

    set starttime [getclock]

    foreach type $pdu_types {
        set count$type 0
    }

    # Loop for interval seconds

    while { [expr [set now [getclock]] - $starttime ] < $interval } {

        set pdu [pduread]

        # filter out unwanted hosts and increment type counters

        if { ( ($host == "all") && ([lindex $pdu 0] != "x") ) || \
            ($hostid == [lindex $pdu 1]) } {
            set typeval [lindex $pdu 0]
            incr count$typeval
        }
    }

    # After interval seconds, calculate average rate over interval

    foreach type $pdu_types {
        set count count$type
        set rate$type [expr [set $count] / double($interval)]
    }

    # Update active charts

    getXYVal_proc
}

# End do forever

##### End main script dis.tcl

# File:      stripchart.tcl
# Contents:  xygraph stripchart library
# System:
# Created:   21-Jan-1994
# Author:    lbob

```

```

# Modified: Aug-1994, LT Mitch Turner

# COPYRIGHT 1994 BBN Systems and Technologies
# 10 Moulton Street      Cambridge, Ma. 02138      617-873-3000
#

# Load help system and files

source "stripchart_hlp.tcl"

global base_help_files
set base_help_files "stripchart1.hlp"

global more_help_file_names
set more_help_file_names "stripchart2.hlp"

#####
# Procedure:  assoc_up_hook
# Description: procedure to initialize new x/y data value lists
# Requires:   nothing
# Arguments:  list of type numbers to create lists for
# Returns:    nothing
# Sideeffects: global lists are created
# Called by:  first_time_assoc_up_hook, StartupGGM, new_chart
#####
proc assoc_up_hook {type_list} {

    foreach type $type_list {
        set Xset X_$type; global $Xset; set $Xset ""
        set Yset Y_$type; global $Yset; set $Yset ""
    }

}

#####
# Procedure:  mark
# Description: procedure to write next line segment or mark for display
# Requires:   xygraph window "w" setup; Note: for the DIS PDU Monitor,
#             this procedure was modified to remove the check to see if
#             window w exists, since this is checked in getXYVal_proc.
# Arguments:  window, tag string, set of X values, set of Y values
# Returns:    nothing
# Sideeffects: line "tagID" is updated
# Called by:  getXYVal_proc
#####
proc mark {w tagId Xset Yset} {

    $w insert $tagId -xdata $Xset -ydata $Yset -label "" -linewidth 2
    -color red

}

#####
# Procedure:  StartupGGM
# Description: proc to startup control panel and/or generic graph views

```



```

# Requires:      nothing
# Arguments:     list of graph types to start
# Returns:       nothing
# Sideeffects:   makes and maps control panel and/or stripchart windows
# Called by:     main script, new_chart
#####
proc StartupGGM {type_list} {

    option add *background white

    # Do not use a strict Motif look and feel
    global tk_strictMotif
    set tk_strictMotif 0

    # Check for required procs
    CheckProcedureExistence build_info_hook
    CheckProcedureExistence assoc_up_hook
    CheckProcedureExistence win_size_hook

    global statusMsg
    set statusMsg {Waiting for initialization}

    #display toplevel windows

    if {![wininfo exists .menub]} {

        global timeWindow fixed_window

        #default to Fixed Window of 2 min
        set fixed_window TRUE
        set timeWindow 120

        ShowWindow $type_list

        make_top_menubar
    }

    # Now build the initial chart windows

    build_info_hook $type_list

}
##### End proc StartupGGM

#####
# Procedure:     ShowWindow
# Description:   proc to show control panel and/or build graph frames
# Requires:      nothing
# Arguments:     list of graph types to build frames for
# Returns:       nothing
# Sideeffects:   Window "." and/or graph frames are set up, with no
#               'section's yet
# Called by:     new_chart, StartupGGM
#####
proc ShowWindow {type_list} {

```

```

if {[wininfo exists .menub]} {

    # Window manager configurations
    wm deiconify .
    wm title . "DIS PDU Monitor Control Panel"
    wm sizefrom . ""
    wm maxsize . 1152 900
    wm minsize . 400 50

    # Build a frame for the top menu buttons
    frame .menub -relief raised -borderwidth 1 -background gray90

    message .statusMsg -background {gray90} -foreground {blue} \
        -font {-*-*-r-*--*-120-*} -aspect {1500} -justify {left} \
        -padx {5} -pady {2} -relief {sunken} -text {} \
        -textvariable {statusMsg}

    # pack widget .
    pack append . \
        .menub {top fillx} \
        .statusMsg {top expand fill }

}

# build graph widget frames -- these will get filled from user code
global type_names

foreach type $type_list {
    toplevel .$type
    wm maxsize .$type 1152 900
    wm title .$type "[lindex $type_names [expr $type-1]] PDUs"
    frame .$type.graphWidgetFrame
    pack append .$type .$type.graphWidgetFrame {top frame center expand
fill }
}

}

##### End proc ShowWindow

#####
# Procedure:    first_time_assoc_up_hook
# Description:  Called from main when the graph views come up
# Requires:     nothing
# Arguments:    none
# Returns:      nothing
# Sideeffects:  The user code assoc_up_hook is invoked
# Called by:    main script
#####
proc first_time_assoc_up_hook {} {

    global assoc_is_up pdu_types
    set assoc_is_up 1

    global statusMsg

```

```

set statusMsg {Starting the initial value query}

# prepare for data collection for initial chart types
assoc_up_hook $pdu_types

}

#####
# Procedure:    make_top_menubar
# Description:  Creates the control panel menubar
# Requires:    window . is set up
# Arguments:    none
# Returns:     nothing
# Sideeffects:  A menubar is created
# Called by:   StartupGGM
#####
proc make_top_menubar {} {

    global pdu_types

    # Create the [Help] menu item. Use the hlp system to fill it.
    # The global variable more_help_file_names contains the name of more
    # help files to load.

    menubutton .menub.help -text "Help" \
        -underline 0 -menu .menub.help.menu -background gray90

    menu .menub.help.menu
    pack append .menub .menub.help {right}
    global base_help_files
    global more_help_file_names
    if (![info exists more_help_file_names]) { set more_help_file_names ""
}
    eval HLP_load .menub.help.menu $more_help_file_names $base_help_files

    # Create the [File] menu

    menubutton .menub.file -text "File " \
        -underline 0 -menu .menub.file.menu -background gray90 -underline
0

    pack append .menub .menub.file {left}
    set m [menu .menub.file.menu]
    set n [menu .menub.file.menu.charts]

    $m add command -label "Display Network Parameters" \
        -command {show_params}

    $m add cascade -label "Print Chart" -menu .menub.file.menu.charts

    $n add command -label "Entity State" -command {printchart 1}
    $n add command -label "Fire" -command {printchart 2}
    $n add command -label "Detonation" -command {printchart 3}
    $n add command -label "Collision" -command {printchart 4}
    $n add command -label "Service Request" -command {printchart 5}

```

```

$N add command -label "Resupply Offer" -command {printchart 6}
$N add command -label "Resupply Received" -command {printchart 7}
$N add command -label "Resupply Cancel" -command {printchart 8}
$N add command -label "Repair Complete" -command {printchart 9}
$N add command -label "Repair Response" -command {printchart 10}
$N add command -label "Create Entity" -command {printchart 11}
$N add command -label "Remove Entity" -command {printchart 12}
$N add command -label "Start Resume" -command {printchart 13}
$N add command -label "Stop Freeze" -command {printchart 14}
$N add command -label "Acknowledge" -command {printchart 15}
$N add command -label "Action Request" -command {printchart 16}
$N add command -label "Action Response" -command {printchart 17}
$N add command -label "Data Query" -command {printchart 18}
$N add command -label "Set Data" -command {printchart 19}
$N add command -label "Data" -command {printchart 20}
$N add command -label "Event Report" -command {printchart 21}
$N add command -label "Message" -command {printchart 22}
$N add command -label "Emission" -command {printchart 23}
$N add command -label "Laser" -command {printchart 24}
$N add command -label "Transmitter" -command {printchart 25}
$N add command -label "Signal" -command {printchart 26}
$N add command -label "Receiver" -command {printchart 27}

$m add command -label "Exit Program" \
    -command {exit_program}

# Create the [Timing] menu item

menubutton .menub.times -text "Timing" \
    -underline 0 -menu .menub.times.menu -background gray90 -underline
0
pack append .menub .menub.times {left}

set m [menu .menub.times.menu]

$m add command -label "Change Sampling Interval" \
    -command {get_interval}

$m add command -label "Automatic Chart Time-scaling" \
    -command {
        global fixed_window pdu_types
        set fixed_window FALSE
        foreach type $pdu_types {
            if (![wininfo exists .$type.graphWidgetFrame.graph]) {
                .$type.graphWidgetFrame.graph configure -xmin "" -xmax ""
            }
        }
    }

$m add command -label "Fixed Window Time-scaling" \
    -command {set fixed_window TRUE}

$m add command -label "Fixed Window Size" \
    -command {win_size_hook}

```

```

# Create the [Charts] menu item

menubutton .menub.charts -text "ChartSelect " \
    -underline 0 -menu .menub.charts.menu -background gray90
-underline 0

pack append .menub .menub.charts {left}
set m [menu .menub.charts.menu]

$m add command -label "Entity State" -command {new_chart 1}
$m add command -label "Fire" -command {new_chart 2}
$m add command -label "Detonation" -command {new_chart 3}
$m add command -label "Collision" -command {new_chart 4}
$m add command -label "Service Request" -command {new_chart 5}
$m add command -label "Resupply Offer" -command {new_chart 6}
$m add command -label "Resupply Received" -command {new_chart 7}
$m add command -label "Resupply Cancel" -command {new_chart 8}
$m add command -label "Repair Complete" -command {new_chart 9}
$m add command -label "Repair Response" -command {new_chart 10}
$m add command -label "Create Entity" -command {new_chart 11}
$m add command -label "Remove Entity" -command {new_chart 12}
$m add command -label "Start Resume" -command {new_chart 13}
$m add command -label "Stop Freeze" -command {new_chart 14}
$m add command -label "Acknowledge" -command {new_chart 15}
$m add command -label "Action Request" -command {new_chart 16}
$m add command -label "Action Response" -command {new_chart 17}
$m add command -label "Data Query" -command {new_chart 18}
$m add command -label "Set Data" -command {new_chart 19}
$m add command -label "Data" -command {new_chart 20}
$m add command -label "Event Report" -command {new_chart 21}
$m add command -label "Message" -command {new_chart 22}
$m add command -label "Emission" -command {new_chart 23}
$m add command -label "Laser" -command {new_chart 24}
$m add command -label "Transmitter" -command {new_chart 25}
$m add command -label "Signal" -command {new_chart 26}
$m add command -label "Receiver" -command {new_chart 27}

# Create the [Host] menu item

menubutton .menub.host -text "Host " \
    -underline 0 -menu .menub.host.menu -background gray90 -underline 0

pack append .menub .menub.host {left}
set m [menu .menub.host.menu]

$m add command -label "All hosts" \
    -command {global host statusMsg
        set host "all"
        set statusMsg "Host: $host\nNetwork sampling interval:
$interval sec"
    }
    $m add command -label "Select host" \
        -command {get_host}

```

```

# Put it all together

tk_menuBar .menub .menub.file .menub.times .menub.charts .menub.host
.menub.help
}
##### End proc make_top_menubar

#####
# Procedure:    xMaxTick
# Description:  determines the highest X-axis time tick value
# Requires:    nothing
# Arguments:    current time, window interval
# Returns:     max tick value in time_t format
# Sideeffects:  none
# Called by:   build_info_hook, getXYVal_proc
#####
proc xMaxTick { time window } {

    set maxTick [expr $time+19]
    return [expr $maxTick - int(fmod($maxTick,20))]
}

#####
# Procedure:    win_size_hook
# Description:  makes and maps a popup to allow user to set fixed window
#               time interval
# Requires:    nothing
# Arguments:    none
# Returns:     nothing
# Sideeffects:  sets the window interval global values
# Called by:   user menu choice
#####
proc win_size_hook { } {

    global timeWindow e

    catch {destroy .winSize}
    toplevel .winSize -background grey90
    wm title .winSize "Fixed Time Window Size"
    wm geometry .winSize +130+250

    set ft [frame .winSize.top]

    set l [label .winSize.l -background grey90 -foreground red \
        -text "Size in minutes:"
    ]

    set e [entry .winSize.e -background gray95 -relief sunken ]

    $e insert 0 [expr $timeWindow/60]

    set fb [frame .winSize.bottom]
    set a [button $fb.apply -text "Apply" \
        -command {
            global e timeWindow

```

```

        set timeWindow [$e get]
        set errflag [catch {set timeWindow [expr
int($timeWindow)*60]} ]
        if {($timeWindow < 60) || ($timeWindow > 599940) || ($errflag
!= 0)} {
            set timeWindow 120
            popup_error "Chart time window must be 1 - 9999 minutes"
        } else {
            set fixed_window TRUE
            destroy .winSize
        }
    }
}

set d [button $fb.def -text "Default" \
    -command {global e; $e delete 0 end;$e insert 0 "2"}]
set c [button $fb.cancel -text "Cancel" -command {destroy .winSize}]

pack append $fb $a {left fill} $d {left fill} $c {right fill}

pack append $ft $l {left} $e {left expand fill}

pack append .winSize $ft {top fill} $fb {bottom expand}

tkwait window .winSize

}
##### End proc win_size_hook

#####
# Procedure:    get_host
# Description:  makes and maps a popup to set hostname to monitor
# Requires:    nothing
# Arguments:    none
# Returns:     nothing
# Sideeffects:  sets the host name
# Called by:   user menu choice
#####
proc get_host { } {

    global host e

    catch {destroy .hostName}
    toplevel .hostName -background grey90
    wm title .hostName "Sending Host to monitor"
    wm geometry .hostName +130+250

    set ft [frame .hostName.top]

    set l [label .hostName.l -background grey90 -foreground red \
        -text "Host name:"
    ]

    set e [entry .hostName.e -background gray95 -relief sunken ]

    $e insert 0 $host

```

```

set fb [frame .hostName.bottom]
set a [button $fb.apply -text "Apply" \
    -command {
        global e host hostid
        set host [$e get]
        if {$host != "all"} {set hostid [gethostid $host]}
        if {$hostid == "error"} {
            popup_error "Host not found.  Spelling or network
connection \
                                may be faulty.  If spelling correct, try
another \
                                computer."
                set host all; set hostid 0
        } else {
            global statusMsg
            set statusMsg "Host: $host\nNetwork sampling interval:
$interval sec"
            destroy .hostName
        }
    }]
set d [button $fb.def -text "Default" \
    -command {global e; $e delete 0 end;$e insert 0 "all"}]
set c [button $fb.cancel -text "Cancel" -command {destroy .hostName}]

pack append $fb $a {left fill} $d {left fill} $c {right fill}

pack append $ft $l {left} $e {left expand fill}

pack append .hostName $ft {top fill} $fb {bottom}

tkwait window .hostName
}
##### End proc get_host

#####
# Procedure:    get_interval
# Description:  makes and maps a popup to set the network sampling
#               time interval
# Requires:    nothing
# Arguments:    none
# Returns:     nothing
# Sideeffects:  sets the network sampling global values
# Called by:   user menu choice
#####
proc get_interval { } {

    global interval e

    catch {destroy .intervalTime}
    toplevel .intervalTime -background grey90
    wm title .intervalTime "Network Sampling Interval"
    wm geometry .intervalTime +130+250

    set ft [frame .intervalTime.top]

```



```

set l [label .intervalTime.l -background grey90 -foreground red \
      -text "Sample time (sec):"
]

set e [entry .intervalTime.e -background gray95 -relief sunken ]

$e insert 0 $interval

set fb [frame .intervalTime.bottom]
set a [button $fb.apply -text "Apply" \
      -command {
        global e interval
        set interval [$e get]
        set errflag [catch {set interval [expr int($interval)]}]
        if {($interval < 1) || ($interval > 60) || ($errflag != 0)} {
          set interval 1
          popup_error "Network sampling interval must be 1 - 60
seconds"
        } else {
          global StatusMsg
          set statusMsg "Host: $host\nNetwork sampling interval:
$interval sec"
          destroy .intervalTime
        }
      }]
set d [button $fb.def -text "Default" \
      -command {global e; $e delete 0 end;$e insert 0 "1"}]
set c [button $fb.cancel -text "Cancel" -command {destroy
.intervalTime}]

pack append $fb $a {left fill} $d {left fill} $c {right fill}

pack append $ft $l {left} $e {left expand fill}

pack append .intervalTime $ft {top fill} $fb {bottom}

tkwait window .intervalTime
}

##### End proc get_interval

#####
# Procedure:  getXYVal_proc
# Description: add X/Y pairs to data lists and plot on stripcharts
# Requires:   stripcharts must exist
# Arguments:  none
# Returns:    nothing
# Sideeffects: updates the graphs with the new values
# Called by:  main script
#####

proc getXYVal_proc {} {
  global statusMsg now pdu_types fixed_window timeWindow host interval

```

```

global rate1 rate2 rate3 rate4 rate5 rate6 rate7 rate8 rate9 rate10
global rate11 rate12 rate13 rate14 rate15 rate16 rate17 rate18 rate19
global rate20 rate21 rate22 rate23 rate24 rate25 rate26 rate27

foreach type $pdu_types {

    # Add X/Y data pairs to list

    set Xset X_$type; global $Xset
    lappend $Xset $now

    set Yset Y_$type; global $Yset
    set rate rate$type
    lappend $Yset [set $rate]

    # Trim data lists so graph will scale to recent Y values

    set points [llength [set $Xset]]
    if { ([expr $points - ($timeWindow/$interval)] > int (0)) \
        && ($fixed_window == "TRUE") } {
        set start [expr $points - ($timeWindow/$interval) - 2]
        set end [expr $points - 1]
        set Xdata [lrange [set $Xset] $start $end]
        set Ydata [lrange [set $Yset] $start $end]
    } else {
        set Xdata [set $Xset]
        set Ydata [set $Yset]
    }

    # Check if graph still exists; if not, remove from active types list

    if {[wininfo exists .$type.graphWidgetFrame.graph]} {
        set index [lsearch -exact $pdu_types $type]
        set pdu_types [lreplace $pdu_types $index $index]
        # puts stderr "Window $type invalid, probably destroyed"
        continue
    }

    # Configure each graph

    if {$fixed_window == "TRUE"} {

        set xMax [xMaxTick $now $timeWindow]
        set xMin [expr $xMax - $timeWindow]

        .$type.graphWidgetFrame.graph configure \
            -xmin $xMin -xmax $xMax -ymin "" -ymax "" \
            -title "[fmtclock [getclock]] \
                "%b.%d.%Y %H:%M:%S      PDU Rate = [set $rate]/sec "]" \
            -xlabel "Time ([expr $timeWindow/60] minute window)"

    } else {
        .$type.graphWidgetFrame.graph configure \
            -xmin "" -xmax "" -ymin "" -ymax "" \

```

```

        -title "[fmtclock [getclock] \
                "%b.%d.%Y %H:%M:%S      PDU Rate = [set $rate]/sec "]" \
        -xlabel "Time (window from startup)"

    }

    # Add the line to the graph

    mark .$type.graphWidgetFrame.graph PDUs $Xdata $Ydata
}

update

}

##### End proc getXYVal_proc

#####
# Procedure:      timeformat
# Description:    custom user x axis formatting routine
# Requires:      TclX extensions
# Arguments:      window (not used) time value to format
# Returns:        nothing
# Sideeffects:    none
# Called by:      build_info_hook
#####
proc timeformat { w time } {

    global timeWindow

    if { $timeWindow > 1800 } {
        set res [fmtclock $time " %H%M"]
    } else {
        set res [fmtclock $time " :%M:%S"]
    }
}

#####
# Procedure:      CheckProcedureExistence
# Description:    Checks for a procedure that should have been provided in
#                 User code.
# Requires:      Nothing
# Arguments:      procname - name of the procedure to check for
# Returns:        nothing
# Sideeffects:    none
#####
proc CheckProcedureExistence {procname} {
    if {[info proc $procname] != $procname} {
        puts stderr "Error: Procedure \"$procname\" must exist to run"
    }
}

##### End script file stripchart.tcl

#
# Help Functions for Auxiliary Views

```

```

# See special copyrights
#
# $Header: /nfs/medea/u0/rel5/rcs/AV/displayLibs/av_hlp.tcl,v 1.3
1993/11/17 18:59:07 djw Exp $
#

#-----
# A general purpose hierarchical help system. The general idea
# is to pass the name of a button in w and the names of help
# files in args. These procedures look for the help files using
# the directories listed in the user's ANMCONFDIR environment variable
and,
# if found, proceed to build a heirarchical menu structure using
# cascade menus as required. The help organization is thus
# removed from the program and can be changed simply by editing the
# text files.
#
# Copyright 1993 Paul Amaranth
# Permission to use, copy, modify, and distribute this
# software and its documentation for any purpose and without
# fee is hereby granted, provided that this copyright
# notice appears in all copies. There are no representations about
# the suitability of this software for any purpose. It is provided
# "as is" without express or implied warranty.
#
# Paul Amaranth 6/2/93 V 1.0
# Modified by David Waitzman on the BBN ANM Project 7/29/93
#
# To keep the name space pollution down, all private external
identifiers
# are prefixed with _HLP, and the one public external identifier is
# prefixed with HLP.
#
#
# Global variables used:
#   _HLP_menus      -- a list of the menus to create
#   _HLP_text-nn    -- The text of the help item. nn is a number.
#   _HLP_last_entry - Used for building the menus
#
# Procs:
#   HLP_load        - Load in the help files. Called by application
#                   This is the ONLY proc directly called by the app.
#                   Returns 0 is successful, 1 if an error.
#   _HLP_add_menus  - Add the menu info in the help file into the
#                   internal format
#   _HLP_instantiate_menus - Realize the menu informat as cascade
#                   menus and command buttons.
#   _HLP_rinstantiate_menu - Recursively called form of above.
#   _HLP_Command    - Proc called to display info when a help button is
#                   pushed.
#
# The help file format looks like this:
#   MENU name
# followed by text for the menu item. A heirarchical arrangement is

```

```

# specified by using additional names, e.g. MENU Name1 Name2 would have
# name2 as a subitem on a cascade menu for Name1. If you want to
# include spaces in the menu label, enclose the string in braces {}.
# NOTE: This string is treated as a list by _HLP_add_menus; it should
# be in list format.
#
# The text is saved in global variables with the name _HLP_txt-n.
Single
# menu items are bound to a command button, multiple are assigned a
# cascade menu.
#
# Text is displayed in a top level window called .hlp
#
# You do not have to define intermediate menus, they are automatically
# created as cascade menus.

```

```

proc HLP_load {w args} {
    #-----
    # Look through the ANMCONFDIR and ANMTCLDIR env variables and "." to
see
    # if we can find a help file with the name 'filename'. If so, open it
    # and build the help menu with the text.
    #-----

    global env
    global _HLP_menus

    set _HLP_menus {}
    set text_no 0

    set path {}
    # If no ANMTCLDIR or ANMCONFDIR environment variables, just use .
    if [info exists env(ANMCONFDIR)] { lappend path $env(ANMCONFDIR)}
    if [info exists env(ANMTCLDIR)] { lappend path $env(ANMTCLDIR)}
    lappend path .

    foreach filename $args {
        foreach p $path {
            set t [catch {set fn [open $p/$filename r]]}
            if {!$t} break
        }

        # If unsuccessful, bail out
        if {$t} {
            puts stderr \
                "Could not open help file $filename; it is not in the path: $path"
            exit 1
        }

        while {[gets $fn line] != -1} {
            if {[string range $line 0 3] == "MENU"} {
                incr text_no
                global _HLP_text-$text_no
                set _HLP_text-$text_no {}
                _HLP_add_menus $text_no [lrange $line 1 end]
            }
        }
    }
}

```

```

    } else {
        set _HLP_text-$text_no [lappend _HLP_text-$text_no $line ]
    }
}
close $fn
}

if {$text_no > 0} {_HLP_instantiate_menus $w}
return 0
}
#-----
# Add this, and all intervening names, into the menu list
# The list data structure is
# { level <Button label> {Subitem list} <text no.> }
# Level 0 items are direct descendants of the parent window.
# Either subitem-list or text_no may be null, but not both at once.
#-----
proc _HLP_add_menus {item_no menu_list} {

    global _HLP_menus _HLP_last_entry

    set fixup_items {}

    # Start at the end of the list to allow for fixups
    for {set mn [expr [llength $menu_list]-1]} {$mn > -1} {incr mn -1} {
        set m [lindex $menu_list $mn]
        set mnu_len [llength $_HLP_menus]
        set found 0
        for {set i 0} {$i < $mnu_len} {incr i} {
            set m_item [lindex $_HLP_menus $i]
            if {[lindex $m_item 0] == $mn && [lindex $m_item 1] == $m} {
                set found $i
                break
            }
        }

        if (!$found) {
            # Add item, only if at end of menu list
            if {$mnu_len == 0 || ($mn == ([llength $menu_list]-1))} {
                set tnum $item_no
                set submenu {}
            } else {
                set tnum {}
                set submenu $_HLP_last_entry
            }
            lappend _HLP_menus [list $mn $m $submenu $tnum]
            set _HLP_last_entry [expr [llength $_HLP_menus]-1]
        } else {
            # Found it, if not at end, add in the submenu to the list
            if {$i != ($mnu_len-1)} {
                set mnu_item [lindex $_HLP_menus $i]
                set submenu [lindex $mnu_item 2]
                if {[lsearch $submenu $_HLP_last_entry] == -1} {

```

```

        set submenu [lappend submenu $_HLP_last_entry]
    }
    set mnu_item [lreplace $mnu_item 2 2 $submenu]
    set _HLP_menus [lreplace $_HLP_menus $i $i $mnu_item]
    set _HLP_last_entry $i
}
}
}

#-----
# Take the menu list structure and turn it into a bunch of
# realizable menus. This is the startup proc for the recursive
# process. Look for level 0 entries and fire them off.
#-----
proc _HLP_instantiate_menus {button} {
    global _HLP_menus

    set cntr 1

    foreach mnu $_HLP_menus {
        set mnu_level [lindex $mnu 0]
        set mnu_label [lindex $mnu 1]
        set mnu_submenu [lindex $mnu 2]
        set mnu_textno [lindex $mnu 3]

        if {$mnu_level == 0} {
            if {[llength $mnu_submenu]==0} {
                eval $button add command -label [list $mnu_label] \
                    -command \{_HLP_Command $mnu_textno\}
            } else {
                incr cntr
                $button add cascade -label [list $mnu_label ->] -menu
                $button.$cntr
                menu $button.$cntr
                foreach sm $mnu_submenu { _HLP_rinstantiate_menu $button.$cntr $sm
                }
            }
        }
    }
}

#-----
# Recursive proc to follow the menu tree, instantiating as
# we go
#-----
proc _HLP_rinstantiate_menu { button entry } {
    global _HLP_menus

    set mnu [lindex $_HLP_menus $entry]
    set mnu_level [lindex $mnu 0]
    set mnu_label [lindex $mnu 1]
    set mnu_submenu [lindex $mnu 2]
    set mnu_textno [lindex $mnu 3]

```

```

if {[length $mnu_submenu]==0} {
    eval $button add command -label [list $mnu_label] \
        -command {_HLP_Command $mnu_textno\}
} else {
    $button add cascade -label [list $mnu_label ->] -menu $button.$entry
    menu $button.$entry
    foreach sm $mnu_submenu { _HLP_rinstantiate_menu $button.$entry $sm
    }
}
}

#-----
# We have been called with the number of a help text to display. Put
# it up in a modal dialog and wait for the user to continue.
# If the amount of text is large enough, use scrollbars.
#-----
proc _HLP_Command { help_no } {

    catch {destroy .hlp}
    toplevel .hlp
    wm title .hlp Help
    wm iconname .hlp Help

    frame .hlp.f \
        -relief raised -border 1
    listbox .hlp.f.list \
        -yscroll ".hlp.f.yscroll set" \
        -xscroll ".hlp.f.xscroll set" -relief sunken -setgrid 1
    scrollbar .hlp.f.yscroll \
        -relief sunken -command ".hlp.f.list yview"
    scrollbar .hlp.f.xscroll \
        -relief sunken -command ".hlp.f.list xview" \
        -orient horizontal
    button .hlp.but -text "Close" -command {destroy .hlp} -background cyan

    global _HLP_text-$help_no

    set width 0
    set line_count 0
    set use_x 0
    set use_y 0

    set helptext [eval set _HLP_text-$help_no]
    foreach str $helptext {
        incr line_count
        set sl [string length $str]
        if {$sl > $width} {set width $sl}
        .hlp.f.list insert end $str
    }

    if {$width > 80} {set width 80; set use_x 1}
    if {$line_count > 25} {set line_count 25; set use_y 1}
    if {$width == 0} {

```



```

        set width 30
        .hlp.f.list insert end "No help available"
    )
    if {$line_count == 0} {set line_count 1}
    set geom [format "%dx%d" $width $line_count]

    if {$use_y} {pack append .hlp.f .hlp.f.yscroll {right fillly} }
    if {$use_x} {pack append .hlp.f .hlp.f.xscroll {bottom fillx} }

    pack append .hlp.f .hlp.f.list {expand fill}
    pack append .hlp \
        .hlp.f {top expand fill} \
        .hlp.but {top}

    .hlp.f.list configure -geometry $geom
# tkwait visibility .hlp
grab .hlp
# tkwait window .hlp

}

```

File: stripchart1.hlp

MENU {About DIS PDU Monitor}

This program monitors Distributed Interactive Simulation networks and displays the PDU traffic rates for each PDU type. The program is written using Tcl and the Tk toolkit (created by John K. Ousterhout), and uses the BBN stripchart library and associated Bell Labs xygraph widget.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227.7013.

Bolt, Beranek, and Newman Inc.
 10 Moulton Street
 Cambridge, MA 02138

This program is copyright 1994 by BBN Inc.

DIS PDU Monitor written by LT Mitchell K. R. Turner, U. S. Navy, July 1994.

MENU {How to use this}

This stripchart application graphs the input PDU count of a DIS network. The initial view uses a fixed time window of 2 minutes and a network sampling interval of 1 second (PDU rates are averaged over the sampling rate). The user can:

Print a chart

File > Print Chart

Change the sampling interval	Timing > Change Sampling Int.
Change the chart time window	Timing > Auto or Fixed
Select PDU types to display	ChartSelect
Select a sending host to monitor	Host

MENU File "Display Network Parameters"
Displays parameters in use by the program. Useful for troubleshooting.

MENU File "Print Chart"
Creates a snapshot postscript file of the current graph in the
~mkturner/charts directory called Chart<timestamp>.ps

MENU File "Exit Program"
Exits program. Chart information is lost.

MENU Timing "Change Sampling Interval"
Changes the interval over which PDU's are collected and their
rates calculated. Sampling interval must be from 1 - 60 seconds.
Note: Mouse-click response delay will increase with longer intervals.

MENU Timing "Automatic Chart Time-scaling"
Causes both the X and Y axes (Time and PDU Count) to automatically
scale to display all the collected data points.

MENU Timing "Fixed Window Time-scaling"
Causes the X-axis (Time) to become a fixed interval, so that only
the last interval's worth of data points are displayed. Default window
is 2 minutes.

MENU Timing "Fixed Window Size"
Sets the Fixed Window size in minutes and causes the x-axis scaling
to shift from automatic to fixed window if necessary. Window must
be 1 minute or greater.

MENU ChartSelect
Creates a new charts with the selected PDU type.

MENU Host "All hosts"
Causes program to monitor all hosts sending DIS PDU's with the correct
exercise id. If using a multicast network, the port and group must be
the same as that entered on PDU Monitor startup.

MENU Host "Select host"
Causes program to monitor one host sending DIS PDU's with the correct
exercise id. If using a multicast network, the port and group must be
the same as that entered on PDU Monitor startup. The complete IP
address must be entered if the host is not local. The message window
below the Control Panel menubar will reflect the host being monitored;
check your spelling!

MENU {Additional Information}
Additional information can be found in the man page stripchart (1).

File: stripchart2.hlp

MENU {What this means}

Each graph shows a strip chart of the PDU traffic for the selected network parameters. It tracks the number of PDU's per second versus the current time. If results are not what you expect, check the network parameters to make sure you are monitoring the correct network environment.

APPENDIX D. INSTALLATION INSTRUCTIONS

The PDU Monitor software is available through the World-Wide Web at the Naval Postgraduate School Computer Science Department home page Uniform Resource Locator (URL): "file://taurus.cs.nps.navy.mil/pub/mosaic/nps_mosaic.html." Pointers to the installation instructions below and the software files are in the NPSNET section of the URL page.

DIS PDU Monitor 1.0 - 1 Sep 94

by LT Mitchell Turner, USN
Naval Postgraduate School
Monterey, California

macedoni@cs.nps.navy.mil

1. Introduction

This program monitors Distributed Interactive Simulation networks and displays the PDU traffic rates for each PDU type. The program is written using Tcl and the Tk toolkit (created by John K. Ousterhout), and uses the BBN stripchart library and associated Bell Labs xygraph widget.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227.7013.

Bolt, Beranek, and Newman Inc.
10 Moulton Street
Cambridge, MA 02138

This program is copyright 1994 by BBN Inc.

2. Files

bin.tar.Z - compressed archive of directory containing the binary Tk shell application (netwish), the Tcl script files to run the PDU Monitor and its help system, the text files for the help system, and manual pages for the stripchart library and the xygraph extension for Tk. The netwish binary is compiled for the IRIX 5.2 operating system (see below for instructions to run the monitor).

src.tar.Z - compressed archive of directory containing the files necessary to extend Tcl 7.3/Tk 3.6 to include the netwish capabilities. Includes netwish extension code (netAppInit.c and disnetlib.c), necessary *.h files, compiled DIS network library functions (libdis_client.a), the xygraph extension (graph.c), and TclX extensions needed by the monitor (TclX*.c). These files are useful to see how Tk was extended to build netwish. However, since the network library was compiled under IRIX 5.2, you will not be able to build netwish unless you are running that OS (see below for instructions to build netwish).

thesis.txt.Z - the compressed text of the thesis that this program was created for.

thesis.ps.Z - compressed PostScript file of complete thesis.

3. Running the PDU Monitor

a. Copy the bin.tar.Z file to a local machine. Run uncompress and tar xvf on the file.

b. If you want users to have general access to the netwish shell, you may move the netwish binary to a directory in generic paths.

c. The printchart function in dis.tcl expects a directory called "charts" at the same level as the directory the program is started from. This directory should have write privileges for everyone who may use the PDU Monitor. If you wish to use another directory, you can modify the printchart function in dis.tcl, replacing "../charts/" with any valid path.

d. To run the generic netwish shell, type netwish. To run the PDU Monitor, type "netwish -f dis.tcl". This must be done a host not involved in the DIS simulation that is running.

e. Choose broadcast or multicast depending on the type of DIS simulation you are running. Enter the network exercise ID, and if using multicast, the network port and multicast group address you are using.

f. The Monitor starts with a chart of Entity State PDU rate. The Help menu item on the right side of the control panel describes all the options available.

4. Building netwish

To build netwish on a system other than IRIX 5.2, you must obtain the source code for the DIS network functions. Contact MAJ Mike Macedonia, USA, at macedoni@nps.navy.mil.

a. Obtain Tcl 7.3/Tk 3.6 and install according to the accompanying instructions (the netwish extensions have not been tested with earlier versions of Tcl/Tk, though the graph.c program has). Tcl/Tk can be obtained by anonymous ftp from ftp.cs.berkeley.edu:/ucb/tcl. The latest

version and various extension packages can also be obtained from
harbor.ecn.purdue.edu:/pub/tcl.

b. Copy the src.tar.Z file to a local machine. Run uncompress and tar xvf on the file.

c. Move the *.h, *.a, and *.c files to the Tk3.6 directory.

d. Modify the Tk Makefile as follows:

- you may need to change the CFLAGS to "-O2 -cckr" (we did for IRIX)
- add the following files to OBJS:
 tclXgetdate.o tclXclock.o tclXcnvclock.o tclXutil.o
 graph.o disnetlib.o
- modify "wish:" to the following:
 wish: netAppInit.o libtk.a \$(TCL_BIN_DIR)/libtcl.a libdis_client.a
 \$(CC) \$(CC_SWITCHES) netAppInit.o \$(LIBS) -o netwish

e. Run the Makefile.

f. Move the netwish binary to the desired directory (see para 3.b-3.f above).

LIST OF REFERENCES

Ferguson, P. M., and Heller, D., *Motif Programming Manual*, O'Reilly and Associates, Inc., 1994.

Institute for Simulation and Training, IST-TR-93-11, *Distributed Interactive Simulation Guidance Document [Draft 2.1]*, University of Central Florida, Orlando, Florida, March 1993.

Johnson, E. F., and Reichard, K., "Tickled Pink," *UNIX Review*, v. 12, March 1994.

Johnson, M. A., *The World Modeler: The Nexus between Janus and Battlefield Distributed Simulation - Developmental*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1994.

Macedonia, M. R., and others, *NPSNET: A Network Software Architecture for Large Scale Virtual Worlds*, Naval Postgraduate School, Monterey, California, 1994.

Operational Support Office, *OSO National Systems Modeling and Simulation for Training and Exercise Support* (briefing), Naval Research Lab, Washington, DC, August 1993.

Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Co., 1994.

Reddy, R., *Advanced Distributed Simulation Concept Briefing*, Advanced Research Projects Agency, Arlington, Virginia, November 1992.

Stevens, W. R., *UNIX Network Programming*, PTR Prentice-Hall Inc., 1990.

INITIAL DISTRIBUTION LIST

- | | | |
|-----|---|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 52
Naval Postgraduate School
Monterey, California 93943-5101 | 2 |
| 3. | Dr. Michael Zyda, Code CS/Zk
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 4. | Professor David Pratt, Code CS/Pr
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 5. | LCDR Donald Brutzman, Code OR/Br
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 6. | Dr. Dan Boger, Code SM/Bo
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 7. | Professor Gary Porter, Code CC/Po
Naval Postgraduate School
Monterey, California 93943-5000 | 2 |
| 8. | MAJ Michael Macedonia, Code CS/PHD
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 9. | Space Systems Academic Group, Code SP
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 10. | Commander, Naval Space Command
ATTN: N112
5280 4th Street
Dahlgren, Virginia 2248-5300 | 1 |
| 11. | LT Mitchell Turner
P. O. Box 57
Fort Belvoir, Virginia 22060-0057 | 2 |

- | | | |
|-----|---|---|
| 12. | Professor Michael Bailey, OR/Ba
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |
| 13. | Professor William Kemple, OR/Ke
Naval Postgraduate School
Monterey, California 93943-5000 | 1 |