NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A RAPIDLY RECONFIGURABLE, APPLICATION LAYER, VIRTUAL ENVIRONMENT NETWORK PROTOCOL

by

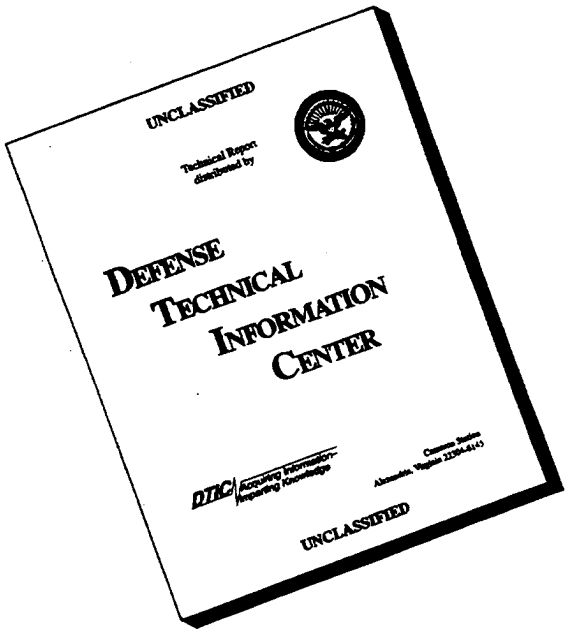
Steven Walter Stone

June 1996

Thesis Co-Advisors: Michael Zyda
John Falby

Approved for public release; distribution is unlimited.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1996	Master's Thesis

4. TITLE AND SUBTITLE
A RAPIDLY RECONFIGURABLE, APPLICATION LAYER, VIRTUAL ENVIRONMENT NETWORK PROTOCOL

5. FUNDING NUMBERS

6. AUTHOR(S)
Stone, Steven Walter

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 United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The current Distributed Interactive Simulations (DIS) Protocol has a limited ability to support real-time, simulated engagements of more than 1000 entities because of its excessive use of network resources. It also lacks the extensibility to add new protocol data units to support new simulation requirements. To solve these problems it is necessary to design and implement a rapidly reconfigurable network protocol that can be easily changed and distributed to all entities in a large-scale simulation. This protocol must be highly flexible and allow for the optimization of data content during execution.

The approach used was to design and build a rapidly reconfigurable network protocol and the tools necessary to use it. This was accomplished in four phases. First, a protocol using the concepts of Self-defined Messages with Multiple Presentations was developed. Second, a formal grammar to describe the protocol was designed. Third, an existing protocol development tool, the DIS Protocol Support Utility, was modified to use the new protocol and grammar. Fourth, the protocol was tested to determine its effect on network resource utilization.

As a result of this effort, a network protocol for distributed simulations that can be optimized at run-time and easily modified has been developed. Testing shows that the protocol can reduce the network bandwidth necessary for a large-scale distributed simulation by up to 70%.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Distributed Interactive simulations, networks, communications protocols	124
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

A RAPIDLY RECONFIGURABLE, APPLICATION LAYER, VIRTUAL ENVIRONMENT NETWORK PROTOCOL

Steven Walter Stone
Captain, United States Army
B.S., United States Military Academy, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

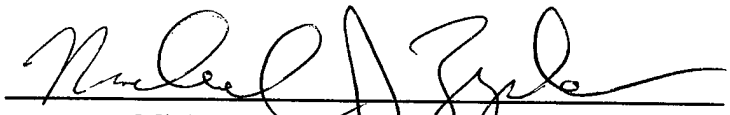
from the

NAVAL POSTGRADUATE SCHOOL

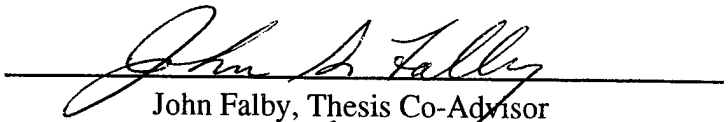
June 1996

Author: 

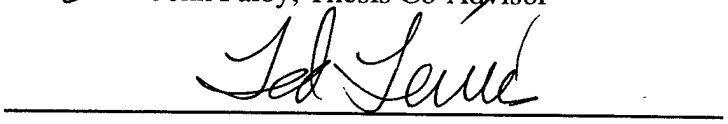
Steven Walter Stone

Approved by: 

Michael Zyda, Thesis Co-Advisor



John Falby, Thesis Co-Advisor



Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The current Distributed Interactive Simulations (DIS) Protocol has a limited ability to support real-time, simulated engagements of more than 1000 entities because of its excessive use of network resources. It also lacks the extensibility to add new protocol data units to support new simulation requirements. To solve these problems it is necessary to design and implement a rapidly reconfigurable network protocol that can be easily changed and distributed to all entities in a large-scale simulation. This protocol must be highly flexible and allow for the optimization of data content during execution.

The approach used was to design and build a rapidly reconfigurable network protocol and the tools necessary to use it. This was accomplished in four phases. First, a protocol using the concepts of Self-defined Messages with Multiple Presentations was developed. Second, a formal grammar to describe the protocol was designed. Third, an existing protocol development tool, the DIS Protocol Support Utility, was modified to use the new protocol and grammar. Fourth, the protocol was tested to determine its effect on network resource utilization.

As a result of this effort, a network protocol for distributed simulations that can be optimized at run-time and easily modified has been developed. Testing shows that the protocol can reduce the network bandwidth necessary for a large-scale distributed simulation by up to 70%.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

As military weaponry has become more complex the cost of operating that equipment has increased dramatically. This rise in cost has necessitated new and less costly methods for training military forces. As the capabilities of computer technology have grown, digital simulations have become a viable method of training. Early attempts focused on stand alone simulators for the most costly weapons systems. Flight simulators for military aircraft were designed to emulate form, fit and function of the aircraft and have become an integral part of training military pilots. Stand alone simulators have also been developed for ground combat systems. The Army's Unit Conduct of Fire Trainer has been in use for over ten years for training gunnery skills of tank crews. These stand alone simulators have proven invaluable and have been credited with some of the success of the recent military operations of the United States. [MAC95a]

However, wars are not fought by a single vehicle. A single tank is not sent into battle. Military forces are organized into units and employed as a composite mass of personnel, weapons and supplies. Units are task organized to meet the needs of the commander. Air, ground and sea weapons systems are combined to provide the commander with all of the capabilities needed to fight and win the battle. The diversity, size and complexity of such a force makes it nearly impossible to accurately model it on a single simulator.

A solution to this problem has been the development of distributed, networked simulations. This technique uses the power of many stand alone simulators and networks them into a larger more complex environment that is capable of accurately simulating all of the associated weapons systems. In this manner, the complexity of the overall model is spread among distributed systems. This approach serves as the basis for the Distributed Interactive Simulations (DIS) effort. [MAC95a]

B. MOTIVATION

As Distributed Interactive Simulations have evolved, so has the requirement for a robust, flexible network communications protocol that is capable of meeting the growing needs of the virtual environment community. These network protocols have evolved from locally created protocols to the SIMNET protocol and now to the Distributed Interactive Simulation (DIS) Protocol as specified by the IEEE 1278 standard [MAC95a]. While the DIS standard has been a great leap forward in specifying networked virtual environment communications, it has not been able to keep up with the rapidly changing uses for distributed virtual environments. At best the DIS standard has been a compromise between the competing demands of distributed simulations [MAC95a].

As simulations have become more and more popular for training military forces, the capabilities demanded of them have increased greatly. Initially used for training single crews, simulators are now regularly used for training small units and there is an increasing demand for distributed simulators capable of simulating large units of over one-thousand systems. Unfortunately, the communications protocols developed for the initial distributed simulations do not scale up well and lack the flexibility necessary for large-scale simulations.

As the use of distributed simulations grow, the need for a flexible, reconfigurable network protocol becomes more apparent. The motivation for this work is to implement a rapidly reconfigurable, application level protocol for use in distributed simulations and to use this protocol in experimenting with large scale distributed simulations.

C. SUMMARY OF CHAPTERS

Chapter II provides an overview of the development of the DIS architecture and communications protocol in term of its history, purpose, objectives and directions. Chapter III outlines the current limitations with regard to large-scale simulations of the DIS

architecture and protocol. Chapter IV presents an overview of previous work in the area of developing a reconfigurable network protocol. Chapter V discusses the design and implementation of the rapidly reconfigurable protocol. Chapter VI contains the results of experiments conducted with the reconfigurable protocol in implementing a large-scale simulation. Finally, Chapter VII provides a summary of findings, conclusions and recommendations for future research in this area.

Appendix A contains the instructions for using the Rapidly Reconfigurable Protocol within NPSNET. Appendix B contains the source listing for the Lexical Analyzer implementation. Appendix C contains the source code generated by the Protocol Support Utility for the Entity State Protocol Data Unit.

II. SIMNET AND DISTRIBUTED INTERACTIVE SIMULATIONS

A. OVERVIEW

The Simulator Networking (SIMNET) and Distributed Interactive Simulations (DIS) protocols are the most advanced efforts to develop distributed virtual worlds. This chapter covers the development of the SIMNET protocol, the relationship between SIMNET and DIS and the DIS protocol.

B. SIMNET

From 1983 to 1989 the Advanced Research Projects Agency (ARPA) developed and demonstrated the technology necessary to link a large number of manned simulators using a local area network. Their goal was to develop a realistic simulation network that could be used to train large numbers of combat vehicle crews in a simulated combat environment. The result of their efforts was SIMNET, at the time the largest communications network in any virtual environment. SIMNET allowed up to 300 combat vehicle crews to engage in mock battle on a simulated battlefield in real time. This was the first successful large networked virtual environment and is currently in use by the United States Army.

The major components of SIMNET are stand alone vehicle mock ups containing a host computer and image generator, and a terrain database. The vehicle mock ups are networked using an Ethernet LAN. This structure has been adapted by many virtual environment users and is currently used by most DIS applications. SIMNET has three classes of protocols: a simulation protocol that conveys information between entities, a data collection protocol for simulation management, and an association protocol to provide transport and session level services over the Ethernet [MAC95a]. The simulation protocol has a well defined set of Protocol Data Units (PDUs) that are used to transport information between entities in the simulation. The vehicle appearance PDU is used to communicate a vehicle's position, orientation and status. Figure 1 shows the structure of this PDU. Fire, Indirect Fire,

Collision and Impact PDUs are used to identify other common battlefield events [MAC95a].

Field Size (bytes)	Vehicle Appearance PDU Fields	
6	Vehicle ID	Site Host Vehicle
1	Vehicle Class	Tank, Simple, Static, Irrelevant
1	Force ID	Object Type - Distinguished Other
8	Guises	Location - x, y, z
24	World Coordinates	
36	Rotation Matrix	
4	Appearance	
12	Markings	Text Field
4	Timestamp	
32	Capabilities	
2	Engine Speed	
2		Stationary bit and padding
24	Vehicle Appearance Variant	Velocity Vector Turret Azimuth Gun Elevation

Figure 1. SIMNET Vehicle Appearance PDU [IEEE93]

SIMNET uses a clever technique of dead-reckoning to reduce the network bandwidth necessary for the simulation to communicate. Each SIMNET node maintains a ghost model

of every other entity in the simulation and maintains a dead-reckoned location and orientation for each of these entities. The simulator continues to generate the image of each entity based on the dead-reckoned location until an update is received from that entity. At that point the ghost entity is updated and dead-reckoning begins again. Each simulator also contains its own ghost entity and sends out an updated location and orientation when the difference between its actual location and its ghost's dead-reckoned location exceeds a threshold. Updates are also sent every five seconds if the entity has not previously sent an update. This is done so stationary entities remain visible to the other players.

SIMNET is more than an application level protocol. It incorporates different layers of the Open Systems Interconnection (OSI) network model thus making it difficult to use on Local Area Network (LAN) topologies other than Ethernet [MAC95a]. SIMNET does not use the Transmission Control Protocol / Internet Protocol (TCP/IP) suite and sits on top of the device driver/ link layer and requires that the process reading/writing SIMNET packets run with root privilege. SIMNET is also limited to approximately 300 players and could not be easily expanded to handle more. Although SIMNET was a defacto standard, it is an incomplete protocol and the problems listed above lead to the development of the DIS standard[ZES93].

C. DIS PROTOCOL

Development of the DIS protocol was begun in 1989 and was sponsored by the United States Army Simulation, Training and Instrumentation Command (STRICOM), ARPA and the Defense Modelling and Simulation Office (DMSO). DIS was based on SIMNET and was designed as a man-in-the-loop simulation in which participants interact in a shared environment from geographically dispersed sites [MAC95a]. The initial objective of the DIS effort was to develop a standard that provided guidelines for the interoperability of defense simulations. DIS attempts to provide a basis of interoperability for large scale virtual environments using a wide variety of different hardware and software platforms.

DIS has been adopted by the Institute of Electrical and Electronics Engineers (IEEE) as an International Standard (IEEE Std 1278-1993).

While the DIS standard adopted many aspects of the SIMNET protocol including its general principles, terminology and PDU formats, it is intended to overcome the limitations of SIMNET and include packet definitions not found in SIMNET [DURLACH95]. DIS is also designed to use the Defense Simulation Internet (DSI), an ARPA project designed to allow thousands of players to link using real and simulated forces to create a Synthetic Theater of War (STOW) [DURLACH95]. Another difference between SIMNET and DIS is that DIS uses the TCP/IP suite of protocols and is thus an application level protocol which can be used on any network topology that uses TCP/IP.

1. Protocol Data Units

The heart of DIS is a set of protocols that are used to convey messages about entities and events, via a network, among the simulation nodes that are responsible for maintaining the status of the entities in the virtual world [DIS94]. Simulation and event information is conveyed by the twenty-seven PDUs defined by the IEEE 1278 DIS standard. Four of the PDUs are used for sending information about entity interaction. The other twenty-three are used for transmitting information on supporting actions, electronic emanations and simulation control [MAC95a]. DIS PDUs are independent of the network media and network protocols being used to transmit them and can be used on most of the current network topologies.

The PDUs used to transmit information about entity interaction are: the Entity State PDU (ESPDU), the Fire PDU (FPDU), and the Detonation PDU (DPDU). The ESPDU is used to communicate information about an entity's current state, including its position, orientation, velocity and appearance. The ESPDU is the most commonly used PDU and in most instances it dominates the network traffic [MAC94]. The format of the ESPDU is shown in Figure 2. The FPDU contains data on any weapons that are fired or dropped. The

DPDU is sent when a munition detonates or an entity crashes. The actual structure of each PDU is very regimented and is explained in full detail in IEEE 1278.

Field Size (bytes)	Entity State PDU Fields	
12	PDU Header	Protocol Version Exercise ID PDU Type Padding Time Stamp Length in Bytes
6	Entity ID	Site Application Entity
1	Force ID	
1	Number of Articulation Parameters	
8	Entity Type	Entity Kind Domain Country Category Subcategory Specific Extra
8	Alternate Entity Type	Same information as above
12	Linear Velocity	X, Y, & Z (32 bit components)
24	Location	X, Y, & Z (64 bit components)
12	Orientation	Psi, Theta, Phi (32 bit components)
4	Appearance	
40	Dead Reckoning Parameters	Algorithm Other Parameters Entity Linear Acceleration Entity Angular Velocity
12	Entity Markings	
4	Capabilities	32 Boolean Fields
N * 16	Articulation Parameters	Change ID Parameter Type Parameter Value

Figure 2. DIS Entity State PDU [IEEE93]

2. The Future of DIS

The IEEE 1278 standard is still an immature protocol and is very much a work in progress [MAC94]. As the possible uses of Distributed Interactive Simulations grow the mission of the DIS standard is expanding. The DIS standard will be modified to meet new roles and will be redefined to increase efficiency [DIS94]. DIS is being envisioned not only as a tool to establish a synthetic battlefield of distributed simulations, but is also being examined for use in a wide spectrum of applications. DIS is planned to grow to be able to handle simulations with 10,000 to 300,000 players. Although DIS has been made into an international standard, it will have significant problems in meeting these future challenges.

III. PROBLEMS WITH THE DIS PROTOCOL

A. OVERVIEW

Although the current DIS protocol has been a major advancement in network protocols for distributed virtual environments, in its current form it is unable to meet the rapidly growing demands of new applications for Distributed Virtual Environments (DVEs). The basic shortcoming of the current DIS protocol is that it is fixed and cannot be easily modified to meet a new communications requirement. It also suffers from limited capacity and being too large for what it needs to do [DURLACH95]. This section will cover some of the areas in which the DIS protocol is lacking.

B. NATIONAL RESEARCH COUNCIL RECOMMENDATIONS

In 1995 the National Research Council released a report on the current state of virtual reality research that made several recommendations about where the government should allocate resources towards this area. In the chapter on networking virtual environments, the study noted several shortcomings with DIS and made several recommendations for future work. The report concluded that the current DIS standard was not designed for generalized information exchange between large scale virtual environments and the standard is too specific and too complex to be useful to general virtual environment development [DURLACH95]. The report also recommended that a major research initiative be undertaken to look at alternatives to DIS.

> What is needed is a major research initiative to investigate DIS alternatives that allow a generalized exchange of information between the distributed participants of large-scale virtual environments. The new protocol needs to be extensible, a feature that DIS lacks. [DURLACH95]

C. INSTITUTE FOR SIMULATION AND TRAINING GOALS

In the DIS Vision Report prepared by the DIS Steering Committee in May of 1994, the committee identified similar shortcomings in the current protocol definition and set forth

goals to correct these problems. The goals set by the steering committee are to increase the functional areas covered by PDUs by planning for the expansion of numbers and types of PDUs due to changes in military doctrine and application, and expansion of DIS into non-Department of Defense (DOD) applications [DIS94]. Another goal is to balance PDU information content and bandwidth efficiency by streamlining PDUs. This could be done by removing static and infrequently changing information from high frequency PDUs (e.g. ESPDU), and send only dynamic data that has changed since last sent and represent data as compactly as feasible [DIS94]. The committee also stated that "In situations where bandwidth is very limited, use PDU sets that have been optimized for bandwidth efficiency and accept possible lack of data [DIS94]". The last and possibly most important recommendation made by the steering committee is "If feasible, define a tailorable set of such PDUs [DIS94]." Both the National Research Council recommendations and the DIS Steering Committee goals point to the fact that the current DIS standard must be further developed in the areas of flexible information exchange and bandwidth efficiency.

D. AREAS REQUIRING IMPROVEMENT IN DIS

1. Network Bandwidth Requirements

It has been estimated that using current protocols such as SIMNET and DIS, a simulation with 100,000 players would require approximately 375 Mbps of network bandwidth to each computer participating in the simulation [MAC95a]. Since such bandwidth is not likely to be available in the near future, the implementation of large-scale virtual environments will require a communications protocol with a much lower bandwidth requirement. Because DIS is a stateless system and does not utilize servers, all data must be distributed to all entities in the system. When the status of one entity changes, it must send an update, as an ESPDU, to every other entity in the simulation. As such, ESPDUs can account for up to 70% of the network traffic [MAC95a]. The problem with this approach is that most of the data in an ESPDU is constant (i.e. vehicle markings, country of manufacture. etc.) which results in the majority of transmitted data being redundant. It

may be possible to significantly reduce the bandwidth necessary for a simulation by reducing the redundant data transmitted by each entity. Of course it remains to be proven that the redundant data is a large part of the problem because each ESPDU is typically small (approximately 144 bytes) [MAC95a]. Another reason to reduce the bandwidth requirements is to enable the use of DIS by the low bandwidth RF communications used by instrumented ranges. This capability is essential for the insertion of live players into a simulation. Unfortunately, it is not currently possible to easily reconfigure the ESPDU to determine if a smaller PDU will reduce the bandwidth requirements.

2. New Network Technologies

One of the promising developments that may solve the bandwidth problem for large-scale environments is the fielding of new high bandwidth networks such as ATM. However, ATM uses fixed length 53 byte cells. It may be possible to optimize the DIS protocol for ATM by making the PDU conform to the 48 byte payload size of the ATM cell [MAC94]. It has been suggested that much of the ESPDU could be reduced in size by eliminating the redundant information such as the host and site identification which could be extracted from the IP address contained in the frame header [MAC94]. Other possible ways of reducing the ESPDU size are removing the fields for vehicle markings, capabilities, alternate entity type and parts of the entity type such as country. It may also be possible to use a single dead-reckoning algorithm and thus reduce the size of the Dead Reckoning Parameters field of the ESPDU [MAC94]. Again, with the current fixed protocol, research on these ideas is not easily accomplished.

3. Non Department of Defense Use

As distributed virtual environments have evolved so has the number of applications outside of the Department of Defense. Distributed virtual environments are being developed for use in industry, entertainment and education. These uses are far from the original motivation for developing DIS of training large numbers of soldiers in a simulated combat environment. Unfortunately, DIS suffers from its origins as a military training tool.

The current PDUs are designed for military applications. This has hindered the acceptance of DIS by the non-DOD user and the DIS steering committee has stated the making DIS more attractive to the non-DOD user is one of its goals. In order for this to happen there must be an easy method of defining new PDUs to meet the needs of a user.

4. Meeting Differing Demands

The current DIS protocol is a compromise between two different uses of DVE's. While the original intent of training soldiers is still valid, the use of DIS for testing new weapons systems is becoming widely accepted. However, the precision requirements of weapons development and testing are very different than those of combat training. These differing precision requirements have lead to a compromises in the format of the protocol. The use of 64 bit floating point numbers for coordinates is required for weapons testing but is excessive for training applications [MAC94]. Reducing the precision of these numbers could be a method of reducing PDU size.

5. New Requirements for Military Simulations

The requirements for military simulations are also growing and dictating changes in the DIS protocol. Two new areas that DIS does not support well are articulated humans in the DVE and the need to aggregate individual vehicles into units. While it is possible to introduce articulated humans into DVE's using DIS it is very cumbersome. Current research at the Naval Postgraduate School has successfully placed articulated humans into its NPSNET virtual environment using the Dismounted Infantryman software. However, the human motion is partially based on postures stored in the Jack Motion Library from the University of Pennsylvania. The human communicates its basic posture in the ESPDU appearance field. Upon receipt of the ESPDU the application decodes the field, determines the posture and requests the appropriate joint angles from the Jack Motion Library. It is possible to support real time articulations driven by users wearing sensors through the articulation parameters of the ESPDU. However, for a large number of articulation

parameters this is extremely inefficient. To allow for efficient real time articulated humans a new more compact PDU needs to be developed. [PRAT95]

Another new need is to aggregate individual vehicles into a unit and to communicate the tactical state of that unit. Currently there are no PDUs defined to support this.

6. Summary

Solutions to the DIS shortcomings listed above are difficult to achieve because it is very difficult to define a new PDU and reconfigure an application to use this PDU. In the current version of NPSNET all of the functionality to read and write DIS PDUs is fixed in the code and requires a modification of the code and recompilation in order to use a new PDU. This is not a trivial task. What is needed is a method to rapidly redefine a PDU and implement its use without having to modify the application code.

IV. PREVIOUS WORK

A. OVERVIEW

As the shortcomings of the current DIS Standard have become apparent there has been significant work in developing new methods of defining protocols for distributed simulations. Many methods for improving the extensibility and scalability of the DIS Protocol have been proposed and some have been tested. This Chapter outlines some of the more significant ideas and proposals for a more extensible and scalable protocol.

At the 13th and 14th Workshops on Standard for DIS, many proposals for the Next Generation of DIS have been presented. Much of the work has focused on making modifications to the current DIS PDUs to allow for greater flexibility. Some has focused on a totally new method of designing these protocols. Concepts such as PDU Profiles and Optional PDU Fields concentrate on making modifications to the current DIS PDUs. Self-describing PDUs and the DIS Protocol Support Utility specify how to define a protocol using a formal grammar and provide methods for implementing new PDU definitions [COHEN96a, CANTER95].

B. PDU PROFILES

1. Overview

The concept of PDU Profiles was proposed by the Protocol Subgroup of the DIS Field Instrumentation Working Group as a method of expanding the functionality and flexibility of the current DIS Standard [SCHUG95]. The group concluded that "multiple standard PDU formats" or profiles, could accommodate different simulation requirements that cannot be met with one PDU format. A PDU Profile would define each PDU in five to ten different ways through the use of optional ordered sets of fields (See Figure 3). Each field would define a data item and all of its attributes. The profile would attempt to define all of the optional ways to form a PDU and to define all of the optional fields. The format of each of the five to ten PDUs would be fixed. The data that could be included, the location

of the data in the PDU, the representation of the data and the amount of data would all be fixed by the profile. [SCHUG95]

	Entity State PDU Profile					
Profile 1	Field ID	Field Length	Field ID	Field Length	Field ID	Field Length
Profile 2	Field ID	Field Length	Field ID	Field Length	Field ID	Field Length
Profile 3	Field ID	Field Length	Field ID	Field Length	Field ID	Field Length
Profile 4	Field ID	Field Length	Field ID	Field Length	Field ID	Field Length

Figure 3. Sample PDU Profile [SCHUG95]

2. Assessment

The concept of a PDU Profile is a limited method of defining options for PDUs so that they can be modified to meet differing simulations' requirements. This approach adds some flexibility and maintains interoperability with the current standard, but it does very little to fix the other drawbacks in the standard. The total number of PDUs increases drastically, increasing the complexity of programming and maintaining a simulation system. Each of the PDUs still contains redundant information. There may be a small reduction in the size of the PDUs but this benefit is counteracted by the increased number of PDUs. There is no reduction in the number of PDUs that are transmitted. And it is very unlikely that all of the profiles can be pre-defined in a way that meets a large set of simulation requirements. [SCHUG95]

C. OPTIONAL PDU FIELDS

1. Overview

An expansion of the PDU Profile approach is to have user specified option fields in the PDU, where, instead of pre-defining the PDU fields and their contents for all of the options, the users define the fields on an as needed basis. This allows the format of the PDU fields to be specified to meet the requirements of a specific simulation exercise. There are a number of ways that the fields can be specified. [SCHUG95]

2. Existing PDU Field Option

This approach uses existing fields in the current DIS PDUs as Option fields to accommodate more user requirements. The idea is to use an existing field as an Option field and to use other existing fields to contain the optional entity attribute data. An example of this approach would be to use the PDU Header Padding field (See Figure 2) to specify the user definable additional data items to be inserted into the PDU, as well as data items to be deleted from the PDU. An existing field that could be used to transmit the additional data is the Articulation Parameters field in the Entity State PDU. The Number of Articulation Parameters field could be defined by the options field in the header to include the additional data that is not normally in the entity state PDU. [SCHUG95]

This method provides very good compatibility with the current DIS fixed format PDUs. It will accommodate user unique data as long as there is an available field in the PDU where the user defined data can be placed. Unfortunately, this approach does little to reduce the size of the PDUs nor does it reduce the transmission of unnecessary data. The user must transmit all the data defined in the PDU, even if there is no use for that data. Another problem is that there may not be a field available for inserting additional data. [SCHUG95]

3. Insertion of a New Option Field

Inserting a new Options field of 8 bits into the header of existing and future PDUs would allow two hundred and fifty-six options for accommodating additional data. If Option 0 is unused and Option 1 is defined to be the standard PDU without modification, there are two hundred and fifty-four additional PDU formats that can be supported. Options two through ten could be pre-defined PDU profiles with every data element defined as in the PDU Profile section above. The additional options would contain the standard header for that PDU type, but all of the fields after the header are optional and are defined by the users through their own method of field definitions. [SCHUG95]

This method allows for completely user definable PDUs and allows a very wide range of PDU definitions while maintaining some compatibility with the current DIS fixed format PDUs. This approach accommodates PDU Profiles as described above, as well as defining one of the options as an Update PDU, containing only the attributes that have changed. PDUs that are optimized for any application can be defined at the cost of having to define the options before an exercise can begin. There is also the overhead of testing this potentially very large set of options for compatibility among the simulations participating in the exercise. [SCHUG95]

4. Assessment

The concept of Optional PDU Fields expands the flexibility of the current DIS Standard but does very little to improve on the scalability of the protocol. Using these concepts, redundant data is transmitted frequently which unnecessarily uses network resources. These ideas do not provide for an easy method of implementation as each of the above concepts will require a significant amount of application re-coding.

D. SELF-DESCRIBING PDUS

1. Overview

The concept of self-describing PDUs is an extension of the Option Field Method described above. All self-describing PDUs contain an option field and all data elements contained in a PDU would be optional and are defined on an as needed basis during pre-exercise setup. [SCHUG95]

Prior to a simulation exercise, the participants would agree upon the format and content of the PDUs required for the exercise. A protocol definition would be developed and distributed to all participants. Each participant would use the protocol definition to initialize a set of PDU look-up tables for decoding PDUs during the exercise. Ideally, this initialization would be done using a generic software package that would automate the construction of the PDU decoding tables. [SCHUG95]

During run-time, the PDU contents could be modified to exclude unnecessary data items with the option field telling the recipient what data items are included in the PDU and how to decode the PDU. Even the enumeration and encoding of the data items are left to the user to define. This provides a great amount of flexibility at the cost of additional PDU processing and set up overhead. [SCHUG95]

2. Self-Defined Messages with Multiple Presentations

Danny Cohen and Moshe Kirsh of Perceptronics have proposed and tested a method of constructing a self-describing protocol definition known as Self-Defined Messages with Multiple Presentations (SDM/MP) [COHEN95, COHEN96a, COHEN96b]. SDM/MP is designed to allow more effective use of communications resources by deferring protocol bindings until exercise set-up time and run time, providing dynamic trade-offs between network bandwidth and data precision [COHEN95]. SDM/MP provides a large amount of flexibility by providing a mechanism for defining new PDUs. SDM/MP is not concerned with the format of specific messages but with how to define the messages easily. SDM/MP provides a framework for the design and implementation of different protocols using a set

grammar and by sharing items from dictionaries. Three components are used to build a protocol: the Data Item Dictionary (DID) that defines the available Data Items and maps those Data Items to classes, the Presentations Dictionary (PD) that defines the possible presentations for each class, and an Exercise Profile that defines a set of messages (PDUs) as a set of Data Items with their allowed presentations. The use of the Data Item Dictionary and the Presentations Dictionary provides a high degree of flexibility while the Exercise Profile constrains that flexibility by selecting only a subset of the possible DID+PD combinations. [COHEN95]

3. Self-Defined Messages

The need to scale up the number of entities in a simulation requires that the communications protocol be as efficient as possible. For a communications protocol to be efficient it must use some type of compression [COHEN96a]. The Self-Defined Message (SDM) part of SDM/MP provides some of this compression by allowing all data elements in a PDU to be optional and transmitted only when needed, for example, a resting entity need not send acceleration. SDM defers the decision of which data elements should be included in the PDU until run time. [COHEN95]

This concept has been used previously in protocols like the Simple Mail Transfer Protocol (SMTP). The header of an SMTP message is a sequence of fields, each starting with a field-ID, followed by a colon, followed by text to the end of the line. The body, separated from the header by a blank line, is random text, terminated by a line with only a period on it. The header is an example of a self-describing message. The message indicates its content by specifying which fields are included. Field-IDs (e.g. "To", "From", "CC", and "Subject") indicate which fields are included. [COHEN95] By allowing data items to be optional and excluding those data items that are not necessary from PDUs, SDM allows the user to send the minimal size PDU possible without transmitting any redundant or unnecessary information.

4. Multiple Presentations

The Multiple Presentations (MP) portion of SDM/MP provides multiple representations of the various classes of data elements. MP offers the user choices between machine and human readability and trade-offs between precision and network bandwidth that may be deferred until exercise setup time and run time. In the current DIS Standard, the single presentation for each data element is designed to accommodate the worst case of both dynamic range and precision. Since only a small fraction of all cases are the worst case, great savings are achieved by providing dynamic choice of data representations depending on the run time situation. SDM/MP provides dynamic choice of data element representation through the list of presentations defined in the Exercise Profile and the Presentations Dictionary. [COHEN95]

Combining Self-Defined Messages with Multiple Presentations creates a domain independent formal method of defining the structure and the presentations of a protocol. It supports the deferral of the choice of the PDU structure to exercise set up and run times and allows both static and dynamic optimizations of the protocol structure. SDM/MP defines the syntax and provides the framework to define the semantics of the protocol, but defines neither the semantics nor the dynamic behavior of the protocol. This is left for the simulation designer. [COHEN95]

5. Dictionaries

The SDM/MP concept is based on a coupled pair of dictionaries, the Data Item Dictionary (DID) and the Presentations Dictionary (PD). The purpose of these dictionaries is to define data elements and their presentations. The DID defines the Data Items and maps them to classes. The PD defines the allowable presentations for each of the classes.

a. Data Item Dictionary

Different Data Items in the DID may be instances of the same class from the PD. The format of the DID is designed to promote human understanding. An example of a DID is contained in Figure 4. The DID lists the Data Item name, the associated class name and

a text field for the definition of the semantics for the Data Item. The Data Item name is used in a message to identify the class name, which is then used to look up the entry in the Presentation Dictionary to find the Data Items presentation.

# Data Item Name	Class Name	Semantics Definition
# ----------	----------	----------
ProtocolVersion	Identifier	# .....
MessageType	Identifier	# .....
TimeStamp	Time	
MessageSerialNumber	Identifier	
SenderEntityID	EntityID	
TargetEntityID	EntityID	
SenderLocation	Location	
TargetLocation	Location	
CenterOfExercise	Location	
SenderLinearVelocity	LinearVelocity	
TargetLinearVelocity	LinearVelocity	
SenderOrientation	Orientation	
TargetOrientation	Orientation	

Figure 4. Data Item Dictionary [COHEN96b]

b. Presentation Dictionary

The Presentation Dictionary maps the classes contained in the DID to their allowable presentations. The PD includes the class name, presentation name, data type, units and description that includes attributes such as range, precision, errors and semantics. For classes that have Multiple Presentations, one presentation is chosen to be the canonical representation. The internal representation of the data should use the canonical presentation. The canonical presentation is marked in the dictionary by having the "*" sign preceding the Presentation name. Figure 5 shows a section of a PD developed for a DIS exercise.

# Class # Name # -------	Name -------	Presentation Data Type ------------	Units -------	Description ---------------
Identifier	*P1	u_char		
	P2	u_short		# Unique Entity ID
	P3	u_short[3]		# Site, Appl., ID
Time	*P1	u_long	[mSec]	# since start of exercise
	P2	u_long	[sec]	# since 00:00:00 1-1-70
	P3	u_short	[min]	# since start of exercise
Location	*P1	double[3]	[m]	# X, Y, Z geo
	P2	float[3]	[m]	# X, Y, Z geo
	P3	float[2]	[deg]	# Long / Lat
	P4	short[3]	[m]	# dX, dY, dZ Ref to sender
	P5	short[2]	[deg]	# dLong / dLat Ref to sender
Linear Velocity	*P1	double[3]	[m/sec]	# Vx, Vy, Vz
	P2	float[3]	[m/sec]	# Vx, Vy, Vz
	P3	float[3]	[rad/sec]	# Vlong / Vlat
	P4	null		# no movement

ArtParams	*P1	struct
		{
		u_char
		u_char
		u_short
		u_long
		char[8]
		}
				
				# Param Type Designator
				# Change Designator
				# ID attatched to
				# Param Type
				# Param Value

Figure 5. Presentation Dictionary [COHEN96b]

c. Exercise Profile

The Exercise Profile specifies the set of allowable messages and their structures, i.e. which Data Items should be included and using presentations. While the DID and PD provide a high degree of flexibility by allowing messages to include any subset of Data Items at any presentation, the Exercise Profile removes much of this flexibility by selecting only a subset of the possible DID + PD combinations. The Exercise Profile specifies everything that may have programming implications regarding the Data Items and their

presentations by means of Message Templates. The Exercise Profile consists of a Profile Header and a set of Message Templates. An example of an Exercise Profile is shown in Figure 6. [COHEN96b]

The Profile Header includes general information common to all of the Message Templates required for encoding and decoding all messages, such as the Profile ID, Presentation Dictionary ID, Data Item Dictionary ID, size of Message ID Type and alignment information. The Profile Header also contains the specification for the TAGS and FLAGS that are used to specify which optional data items are present in a message. The Tag Organization, Submask Type, Submask Continuation and Submask Bit Alignment fields all relate to the TAGS and FLAGS that are used to specify the optional data items. These fields and their format will be discussed in a later section. [COHEN96b]

d. Message Template

The Message Template (MT) is a mechanism to specify the possible content of various messages (PDUs). MTs specify what are the mandatory Data Items, optional Data Items, and their allowed presentations. An example of a Message Template is in Figure 7. The MT also specifies the mechanisms for the format and layout of each Data Item. Within the Exercise Profile, Message Templates are enclosed by curly braces { } and are separated by commas. The first item in a Message Template is a Message Header which contains the message type specification and may contain attributes such as Alignment Policy and Submask that overwrite the general attributes from the Profile Header. Messages may contain any type of Data Items specified in the DID. These Data Items and their presentations are further specified in Data Item Templates. [COHEN96b]

```
<                                                    # begin of profile
        # Header
        Profile-Version          1.0,
        Profile_ID               profile,
        PD_ID                    DIS3_PD,
        DID_ID                   DIS3_DID,
        Msg_ID_Type              u_char,
        Tag_Organization         Grouped,
        Alignment_Policy         natural,
        Submask_Type             u_char,
        Submask_Continuation     YES,
        Submask_Bit_Alignment    NO,

        # Message                Item Templates
        # Template Type           |
           |                      |
           V                      V
        { 1,                                     # Message Template 1
                                                 # Message Header
        # .....
                                                 # Message Items
          ProtocolVersion( P1 ),                 # u_char      Identifier
          SenderID( P2 ),                        # u_short     EntityID
          TimeStamp( P1 ),                       # u_long      [mSec]
          [ SenderLocation( P1,                  # double[3]   X, Y, Z geo [m]
                            P2 )  TAG],          # float[3]    X, Y, Z geo [m]
          [ SenderLinearVelocity( P1 ) TAG],     # double[3]   Vx, Vy, Vz
                                                 #             [m/sec]
          [ SenderOrientation( P2 ) TAG],        # char[3]     Psi, Theta, Phi
                                                 #             [rad]
          [TargetID( P1 ) TAG],
          [ TargetLocation( P1,                  # double[3]   X, Y, Z geo [m]
                            P2 ) TAG],           # float[3]    X, Y, Z geo [m]
          [ TargetLinearVelocity( P1 ) TAG],     # float[3]    Vx, Vy, Vz [m/sec]
          [ TargetOrientation( P2 ) TAG]         # char[3]     Psi, Theta, Phi
                                                 #             [rad]
        },

        { 2,                                     # Message Template 2
          ProtocolVersion( P1 ),                 # u_char      ID
          SenderID( P2 ),                        # u_short     UEID
          [ TimeStamp( P1, P2 ) FLAG],           # u_long      [mSec]
          [ CenterOfExercise( P1 ) TAG],         # double[3]   X, Y, Z geo [m]
        }

>                                                # end of profile
```

Figure 6. Exercise Profile [COHEN96b]

```
{  2,                                    # Message Template 2
   ProtocolVersion( P1),                 # u_char     ID
   SenderID( P2 ),                       # u_short    UEID
   [ TimeStamp( P1,                      # u_long     [mSec]
                P2 ) FLAG ],             # u_long     [25mSec]
   [ CenterOfExercise( P1 ) TAG ],       # float[3]   X, Y, Z geo [m]
}
```

Figure 7. Message Template [COHEN96b]

e. Data Item Template

The Data Item Template (Figure 8) is a mechanism to specify how the information for a particular Data Item is presented. All names in the Data Item Template are referenced in the DID and PD. Class names and item names are referenced in the DID and PD, and all Class and presentation names are referenced in the PD. There are two types of Data Item Templates: Mandatory and Optional. Mandatory Data Items must appear in every message of a given type. Optional Data Items are marked in one of two ways, depending on their presentation. Each Data Item Template contains a Data Item Name and a list of allowable presentations. Optional Data Items are enclosed by a pair of square brackets "[ ]" and contain a marking field, TAG or FLAG, that specifies how the presence of the Data Item is indicated in a message. Marking the existence of Optional Data Items by a FLAG is suitable for Data Items that have a high probability of appearance and a small number of presentations. A bit-mask is used to keep the flags for all optional Data Items of this type. Because reserving bits in the bit-mask may become expensive for a message with a large number of optional data items that have a low probability of inclusion, a TAG that indicates the presence of a Data Item can also be used. [COHEN96b]

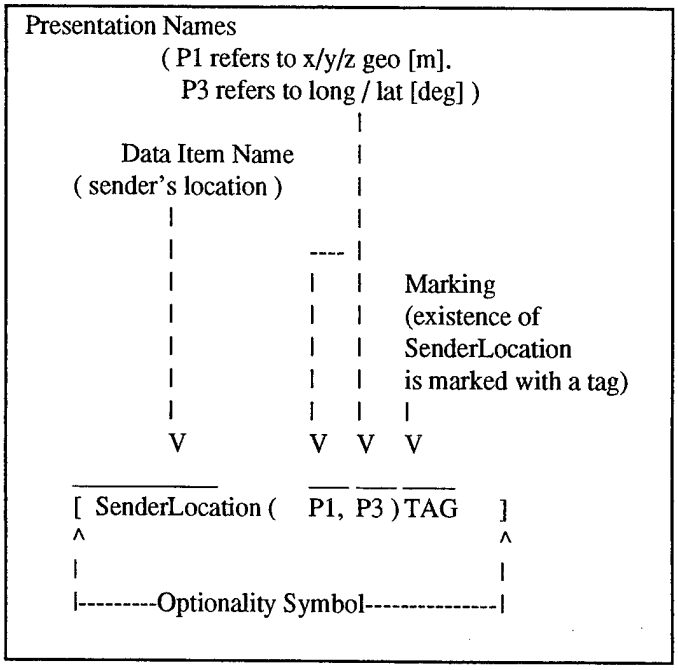


Figure 8. Data Item Template [COHEN96b]

6. Advantages of SDM/MP

The primary advantage of the SDM/MP method of specifying a protocol definition is that the final message structure is not defined until the application is running. Unlike the current DIS protocol, where the final message structure is defined by a committee long before an exercise is run, SDM/MP defers the final binding until exercise setup time and then run time. At exercise setup time, an Exercise Profile with the proper Data Item Dictionary and Presentation Dictionary is selected. The selection of the Exercise Profile binds the messages, data elements, and presentations. The final binding is done at execution time when messages and their format are selected dynamically by the application. The application selects the proper message type and the best format from the Exercise Profile depending on the status of the application at the time. [COHEN95]

This ability to defer the selection of the message type and format until run time gives the application the ability to optimize its communications while running. This gives the simulation a great amount of flexibility that will help greatly in increasing the

scalability and extensibility of the communications protocol. This flexibility comes at the expense of added overhead at exercise setup time and increased message processing at run time. However, experimentation by Cohen and Kirsh has shown that SDM/MP reduced the communications requirements by factors of 4X to 40X, depending on the nature of the exercise [COHEN96a]. [COHEN96b]

E. PROTOCOL SUPPORT UTILITY

1. Overview

The Protocol Support Utility was developed by Major Michael Canterbury at the Naval Postgraduate School as an aid to rapid development of network protocols for distributed simulations. It was designed as a tool to help implement solutions to many of the problems with the current DIS Protocol. "The Protocol Support Utility provides a means to easily manipulate the form and content of a given PDU, and automatically generate the program source code necessary to implement any changes made to the DIS Protocol." [CANTER95].

The Protocol Support Utility uses a modified Bachus-Naur Form (BNF) grammar to describe the protocol and allows the user to edit the grammar specification of the protocol. When the protocol specification is complete, the Protocol Support Utility will generate C++ source code that reads and writes the specified protocol. The Protocol Support Utility uses a lexical analyzer to read the protocol specification. The lexical analyzer uses code generated by the UNIX utility program LEX. The Protocol Support Utility uses the concept of an Application Profile to generate the source code needed for a given application. The Application Profile is a collection of functions needed to produce the desired source code for a given system. Currently, the Protocol Support Utility supports two application profiles; the NPSNET Profile to support the Naval Postgraduate School's NPSNET Virtual Environment and the Class-based Profile which was used to demonstrate the ability of the Utility to generate source code based on different programming paradigms. [CANTER95]

2. Protocol Support Utility Grammar

The formal grammar used by the Protocol Support Utility is a modified Bachus-Naur Form grammar. BNF is a metalanguage typically used as a formal method for defining programming language constructs. BNF was chosen for this application because of its wide acceptance and its simplicity. BNF may be easily adapted to meet almost any application need and it is the embodiment of simplicity [CANTER95].

The grammar is based upon the fundamental concept of syntactic units and terminal symbols. Syntactic units are the grammar constructs which are considered valid in the particular language being described. In BNF, syntactic units are generally enclosed in angular brackets, ***<grammar construct>***. The terminal symbols used in BNF are the primitive or atomic language elements on which the described grammar is based. [CANTER95]

The syntactic constructs used in the grammar are shown in Figure 9. A pair of angular brackets "<< >>" is used to distinguish PDUs. Braces "{}" are used to signify dynamic data structures and square brackets "[ ]" indicate a sized array of elements. The semi-colon ";" is used to end each definition.

Grammar Construct	Meaning
<< PDU >>	PDU
< composite >	composite data element
{< composite >}	composite structure
< composite > [size]	composite element array
atomic	atomic data element
{atomic}	atomic structure
atomic [size]	atomic element array
enum8, unint16,...	alias for primitive data type
::=	"is defined as"
;	"end definition'

Figure 9. Descriptive Grammar Constructs [CANTER95]

These grammar constructs are used to model the protocol under development. As an example, the current DIS Protocol was modeled using this grammar. The definition of the DIS Protocol was broken into three parts: the Atomic Data Types, the Composite Data Types and the Protocol Data Units. The Atomic Data Types expressed in the grammar are actually aliases for the basic data types found in a typical programming language. For the Protocol Support Utility, the basic data types are those found in the C++ programming language. The correspondence between the Atomic Data Types and the basic C++ data types are shown in Figure 10. Capitalization is used as a means of discriminating which Atomic Data Types were meant to be formal data type definitions. Using this approach, the Atomic Data Type declaration **PDUType** signifies a formal data type definition while the declaration **num_params** indicates an Atomic Data Element defined in terms of some

primitive data type associated with the programming language used for the implementation. [CANTER95]

Grammar Primitive	C++ Data Type
bool32	unsigned int
enum8	unsigned char
enum16	unsigned short
float32	float
float64	double
pad8	char
pad16	short
pad32	unsigned int
uint8	unsigned char
uint16	unsigned short
uint32	unsigned int

Figure 10. Grammar to Data Type Correlation [CANTER95]

These Atomic Data Types are used to build a set of Composite Data Types which are in turn used to build the definition of the PDUs in the protocol. Figure 11 illustrates use of the grammar in modeling a typical PDU, a Composite Data Element and an Atomic Data Element. [CANTER95]

```
<<EntityStatePDU>>::=      <PDUHeader>
                           <EntityID>
                            ForceID
                            num_articulat_params
                           <EntityType>
                           <Alt_EntityType>
                           <VelocityVector>
                           <EntityLocation>
                           <EntityOrientation>
                            entity_appearance
                           <DeadReckonParams>
                           <EntityMarking>
                            capabilities
                           <ArticulaParams>[ART_PARAMS]
                            ;

<PDUHeader>       ::=       protocol_version
                           exercise_id
                           PDUType
                           protocol_family
                           time_stamp
                           length
                           padding16
                           ;

ForceID           ::=       enum8;
```

Figure 11. Protocol Utility Grammar [CANTER95]

Given this structured grammar specification with which to build a protocol definition, it was possible to build tools to manipulate the definition of the protocol by altering its grammar-based description. The modified description may then be used to regenerate the program code needed to implement the protocol. [CANTER95]

3. Protocol Support Utility

As stated earlier, the purpose of the Protocol Support Utility is to provide an automated facility for protocol development, to include authoring, editing, and implementation. To achieve this goal the Protocol Support Utility had to have the following capabilities: read and parse the given grammar specification, maintain and store the grammar in an internal representation suitable for other program use, support both authoring and editing of the protocol grammar as necessary, support source code generation based upon specific Application Profiles, and support interactive use of the tool through a simple to use graphical user interface (GUI) [CANTER95]. The components of the Protocol support Utility are designed to support these functions.

a. Lexical Analysis

The Protocol Support Utility uses a LEX-based lexical analyzer for parsing an input grammar. The analyzer is designed to recognize the unique delimiters which identify PDUs and Composite and Atomic Data Elements. The analyzer is basically a finite state machine with five states. These states are:

<C_CMMT>

<CPP_CMMT>

<INITIAL>

<PDU_DEF>

<STRUCT_DEF>

The first two states are designed to support embedded comments in both the C and C++ conventions. The analyzer detects a comment and ignores subsequent tokens until it recognizes an appropriate comment ending delimiter. The next state, <INITIAL>, is the rest state. In this state, the analyzer is waiting for the definition of a protocol element. This state corresponds to the left side of a production rule and the analyzer will accept any grammar construct. When the analyzer detects a PDU or Composite construct the <PDU_DEF> or <STRUCT_DEF> state is set and the analyzer expects to encounter only

Composite or Atomic constructs. The <PDU_DEF> and <STRUCT_DEF> states correspond to the right-hand side of a typical production rule. [CANTER95]

When the analyzer detects an allowable construct, it will invoke one of the Table Management function associated with the Utility. These functions provide the means to initialize, load, and update the internal data tables (Symbol, PDU, etc.) Through these functions, the internal data tables will be fully loaded when the analyzer detects the end of the grammar file. This lexical analysis process is used repeatedly during the program execution as a method of initializing and updating the grammar stored in the internal data tables. [CANTER95]

b. Internal Data Tables and Structures

The heart of the Protocol Support Utility lies in the data structures used to store and maintain the protocol grammar. These tables serve as the repository for the protocol elements that are defined in a given grammar. Four separate tables are used: the Symbol Table, the PDU Table, the Composite Table and the Atomic Table. The Symbol Table is the structure in which the character strings associated with each grammar are stored. With each of these strings three index fields are also stored. Each index field corresponds to one of the protocol constructs (PDU, Composite, or Atomic) supported by the grammar. Only one of the three index fields is used for any entry. If the character string corresponds to a PDU, only the PDU index field is set, the other two remain undefined. This table provides a way to remember the type of protocol element that is associated with each character string.

Each element in the PDU Table represents a single PDU definition consisting of a table index which points to its name in the symbol table, an element count and an array of protocol elements. The Composite Table is similar to the PDU Table but is used to store the definition of a Composite Data Element.

The Atomic Table is slightly different from the other tables used to store PDU and Composite elements. An entry in the Atomic Table consists of two Symbol Table

indices and a single character string. The first index into the Symbol Table references the position where the name of the Atomic element is stored. The second index provides the name of the primitive data type associated with the Atomic element. The character string in each entry is to hold a string representing a programming language data type during source code generation. [CANTER95]

c. User Interface

The Protocol Support Utility is operated through a Graphical User Interface (GUI) that consists of a Main Program Window which supports a pulldown menu bar, a grammar view window, and a display of the runtime status of the Utility. Also supported are a series of buttons which launch the editors for each type of protocol element. [CANTER95]

d. Grammar Editors

The Utility supports editing of the grammar through an ASCII-based text editor. As currently implemented, the Utility uses **jot,** the default text editor on most Silicon Graphics computers. When the user launches the editor for a given type of protocol element, it reads in the appropriate grammar file and displays it for the user. Editing is done as one would edit any text file. When editing is complete, the user must update the grammar in the internal tables by pushing the "Update" button. "Update" reinitializes the internal tables and invokes the lexical analyzer on the newly defined grammar file. [CANTER95]

e. Source Code Generation

The principle feature of the Protocol Support Utility is the ability to generate program source code based upon a previously parsed grammar [CANTER95]. Because of the diversity of the Distributed Simulations community, it is unlikely that the source code produced for one application would be usable by another application. As such, the Protocol Support Utility uses the concept of an Application Profile to provide flexibility in terms of the structure, content, and scope of the source code produced. An Application Profile is a

collection of functions needed to produce the desired source code for a given system [CANTER95]. Application Profiles may differ greatly. The characteristics of a Profile are dictated by the implementation dependencies within the host system application. Currently, the Protocol Support Utility has incorporated two Application Profiles, the NPSNET Profile and the Class-based Profile.

The NPSNET Profile has been designed to support NPSNET which is a DIS compliant, networked software architecture built to support large scale virtual environments [MAC94]. NPSNET was chosen as a candidate profile because of its local availability and its maturity as a host platform for DIS use. [CANTER95]

The Class-based Profile was designed to demonstrate the ability to generate source code based upon different programming constructs or paradigms. Under this Profile, each PDU is implemented as a derived class.

4. Advantages of the Protocol Support Utility

The Protocol Support Utility provides an easy and flexible method of building a new or modified protocol definition. Its formal grammar provides an unambiguous definition of a protocol that is concise and easily understandable. The ability to rapidly define a protocol and generate appropriate source code is necessary in the effort to develop solutions to the problems with DIS.

If there is an area where the Protocol Support Utility is lacking, it is in defining a flexible protocol. As it operates now, it is very easy to generate a new protocol definition, but that protocol is then fixed. It provides no ability to delete redundant or unnecessary data items at runtime. Nor does it provide any flexibility in the representation of the data elements.

F. SUMMARY

The concepts of PDU Profiles, User Defined Option Fields, Self-describing PDUs and the Protocol Support Utility all contribute to solving the problems of the current DIS Standard. However, none of these, by themselves, allow for a truly extensible and scalable

protocol. What is needed is a protocol that incorporates the best elements of each of these concepts.

V. DESIGN OF A RAPIDLY RECONFIGURABLE PROTOCOL

A. INTRODUCTION

The International Telecommunications Union of the United Nations defines a protocol as "A set of rules and formats (semantic and syntactic) which determines the communication behavior of entities in the performance of functions." [Schug95]. A simulation protocol is the set of rules that specify the services to be performed on an entity and on its attributes over a communications link [Schug95]. The messages used within a simulations protocol typically specify the object, the attributes of the object, and the services to be performed on the object or its attributes. The goal of such a simulation protocol is to allow for interaction between simulation systems that may have been built for separate purposes, or, built using different technology. To do this a simulation protocol must support interoperability between different and geographically dispersed simulation entities, a standard method of simulation applications interfacing, central simulation management in distributed environments, and reduced software development and maintenance [Schug95].

When the DIS Protocol was designed, it met this goal, given the uses of simulation technology at the time. However, the rigid and limited methodology used in the current DIS Protocol does not meet the demands of the current and future uses of distributed simulations. The DIS Community has recognized this and has proposed a number of protocol related improvements that increase the extensibility and scalability of the protocol. Several of these suggested improvements serve as the foundation of this research. They are: increasing the functional areas covered by PDUs, balancing PDU content with bandwidth efficiency, streamlining PDUs, and defining a tailorable set of PDUs [Canter95]. All of these ideas are encompassed in the motivation for this work as stated in Chapter I. "The motivation for this work is to implement a rapidly reconfigurable, application level network protocol for use in distributed simulations and to use this protocol in experimenting with large-scale simulations".

B. DESIGN CRITERIA

A protocol that meets all of the goals listed above must be easily changed and must allow for the rapid development of PDUs to meet new simulation requirements. It must optimize the format of the transmitted data to the minimum number of bits that provide the required precision. The protocol must allow unnecessary or unchanging data to be eliminated from PDUs at run time. But, the protocol must also provide backward compatibility with the DIS 2.03 Standard. And, most important of all, it must allow for all of these changes without requiring code changes to the simulation application. Each of the proposals and efforts covered in Chapter IV add some of the desired functionality to the DIS Protocol. However, none meet all of the goals stated above. To meet these goals, the best features of each idea must be combined into one protocol.

The approach adopted for this thesis effort is to combine the concepts of Self Defined Messages with Multiple Presentations (SDM/MP) and the formal grammar and automated code generation of the Protocol Support Utility. The ideas of SDM/MP provide the ability to optimize the format of the data transmitted to the minimum number of bits required and to eliminate unnecessary or unchanging data from PDUs. The Protocol Support Utility provides the functionality to quickly and easily modify the protocol and to automatically generate the source code necessary to use the newly modified protocol. The strengths of the two approaches compliment each other. Where the SDM/MP approach provides a high degree of flexibility in a protocol, it does not provide for rapid modification of the protocol definition nor for the generation of source code. The Protocol Support Utility allows for rapid protocol modification and source code generation, but its simple grammar specification does not provide any flexibility in the protocol once it is defined. Combining these two approaches will take advantage of the strengths of both and increase the extensibility and scalability of the DIS Protocol.

C. SIMULATION SYSTEM DESIGN

Early in this project, it became very apparent that a simulation application must be designed correctly in order to take advantage of the benefits of a reconfigurable network protocol. As experimentation with the Protocol Support Utility and the Reconfigurable Network Protocol progressed, many difficulties were encountered between the network protocol and the existing simulation application, NPSNET-IV. These difficulties centered around the data representation within NPSNET and with its original design as a DIS compliant simulation system. As work progressed a new, more robust architecture for distributed simulation applications was developed.

1. NPSNET-IV

NPSNET-IV was designed as a DIS compliant, networked software architecture to support large-scale distributed virtual environments [MAC94]. The NPSNET-IV series of software has been an excellent DIS compliant visualization package, however, its limitations have become increasingly obvious. Since its conception, NPSNET-IV was designed to operate strictly in a DIS environment and lacks the flexibility to use any other network protocol. As designed, the NPSNET-IV architecture is monolithic and is not easily modified (Figure 12). Being rooted in DIS has allowed a visualization system which uses the IRIS Performer Graphics Package to be built around a very efficient DIS Networking Library. Because it was not envisioned that NPSNET-IV would be used outside of the DIS world nor that the system would need to be ported to systems other than SGI, the DIS Protocol and IRIS Performer Graphics Library are deeply embedded through out the entire system. [BARKER96]

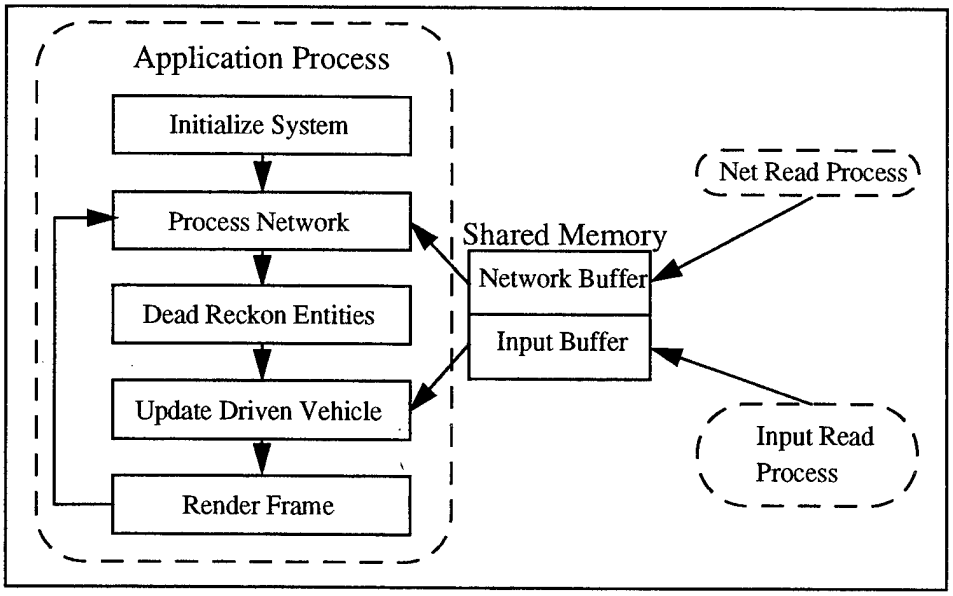


Figure 12. NPSNET-IV Architecture

Because of its design as a DIS compliant system, the logical choice for the internal data representation within NPSNET-IV was the data representation used within the DIS Protocol. As such, every data structure related to DIS PDUs within NPSNET-IV is in the same format as DIS. Similarly, every data element contained in the DIS PDUs processed by NPSNET, Entity State, Fire, and Detonation, is represented as a data structure within the system. This DIS dependent internal data representation has led to difficulties in integrating a reconfigurable network protocol with NPSNET-IV. During the development of the Protocol Support Utility, the lack of a standard data naming convention within the DIS Protocol and NPSNET-IV led to difficulties in creating the NPSNET Application Profile [CANTER95]. Additionally, the dependence of the NPSNET-IV network library on the DIS Protocol definition created problems. In his findings, Canterbury states:

> Without a clearly defined API, automated source code generation is problematic. A more generic network harness should be developed, one not dependent on specific protocol entity definitions. Once implemented, this network harness would serve as the API needed to accommodate automated source code production for DIS applications. [CANTER95]

While Canterbury was able to overcome these difficulties for the DIS Protocol as it is now defined, changing the protocol definition caused problems with NPSNET-IV. Because NPSNET-IV internally expected all of the data contained in a PDU, in exactly the format specified in the PDU, any attempt to change the protocol definition caused the application to fail. An early approach in this thesis effort was to develop a code module that would reside between the Net Read Process and the Network Buffer and translate the data from an incoming PDU into the expected format. This proved to be troublesome and added additional processing to a time-critical process. This approach was quickly abandoned. The solution to these problems is to completely separate the internal data representations from the external communications protocol.

2. Application Program Interface

What was needed was a specific Application Program Interface (API) between the Simulation Protocol and the Simulation Application. APIs are the interface between the Simulation Application Programs and the Simulation Protocol. The API encodes the application program data into service data units which are presented to the simulation protocol. The API translates between the data in the application program , e.g., aircraft location, and the protocol service, e.g., send X to Y. APIs support portability of the application software and of simulation entities. The simulation protocol specifies what the API must provide to interface with the protocol. The system implementor is free to code the API as he sees fit, using any programming language, as long as the API interfaces correctly with the simulation protocol. [SCHUG95]

It is very important to isolate the API from the simulation protocol because the API is unique to every application and system and requires custom coding for each application. Separating the API from the simulation protocol isolates the interface functions that require custom coding from the simulation protocol services that should not change. Figure 13 illustrates the relationship between simulation application programs, APIs and the simulation protocol.

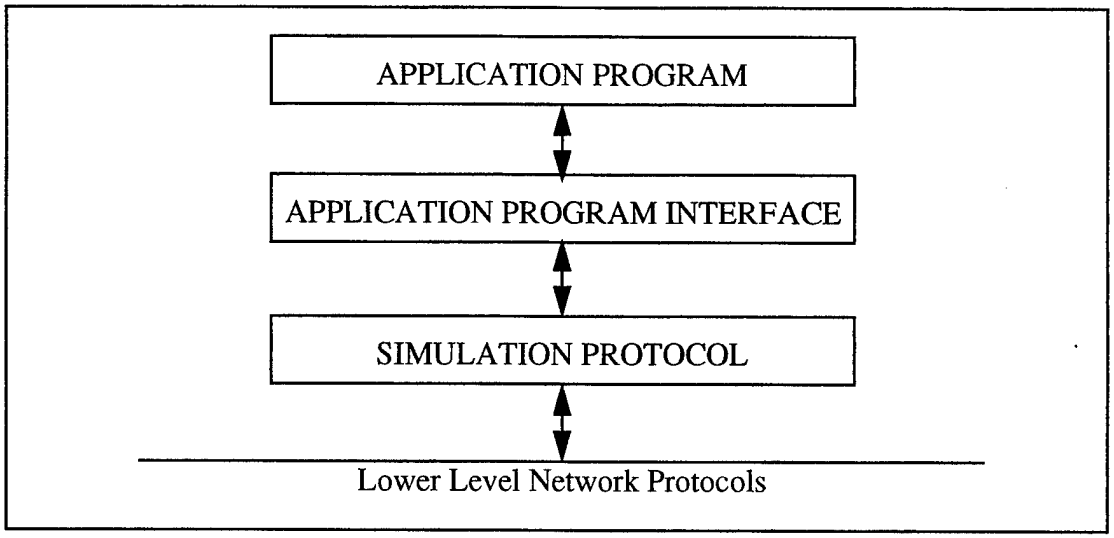


Figure 13. Application Program / API / Simulation Protocol Relationship [SCHUG95]

The concept of an API between the application and the simulation protocol is used in NPSNET-IV. NPSNET-IV has a well defined, efficient DIS network library that performs the API functions between the application and the simulation protocol. The problem with the implementation used is that the internal data representation is the same as the data representation within the protocol, and, there is no provision for any flexibility. Another problem is that the NPSNET-IV network library is dependent on the DIS 2.03 Standard. It cannot be used to process any PDUs other than DIS. This obviously will not work with a reconfigurable simulation protocol.

3. OpenNPSNET Architecture

The OpenNPSNET Architecture was designed by the NPSNET Architecture Design Group to overcome many of the current limitations with distributed virtual environments providing a robust platform for developing large-scale distributed simulations. The OpenNPSNET Architecture is built around a set of software modules called "managers". The managers work independently to manage their part of the system while working together concurrently to accomplish the end goal. The managers are used to

provide networking, graphics, and interface capabilities to be used by a large-scale, distributed visual simulation system. Figure 14 gives a graphical representation of the OpenNPSNET Architecture.

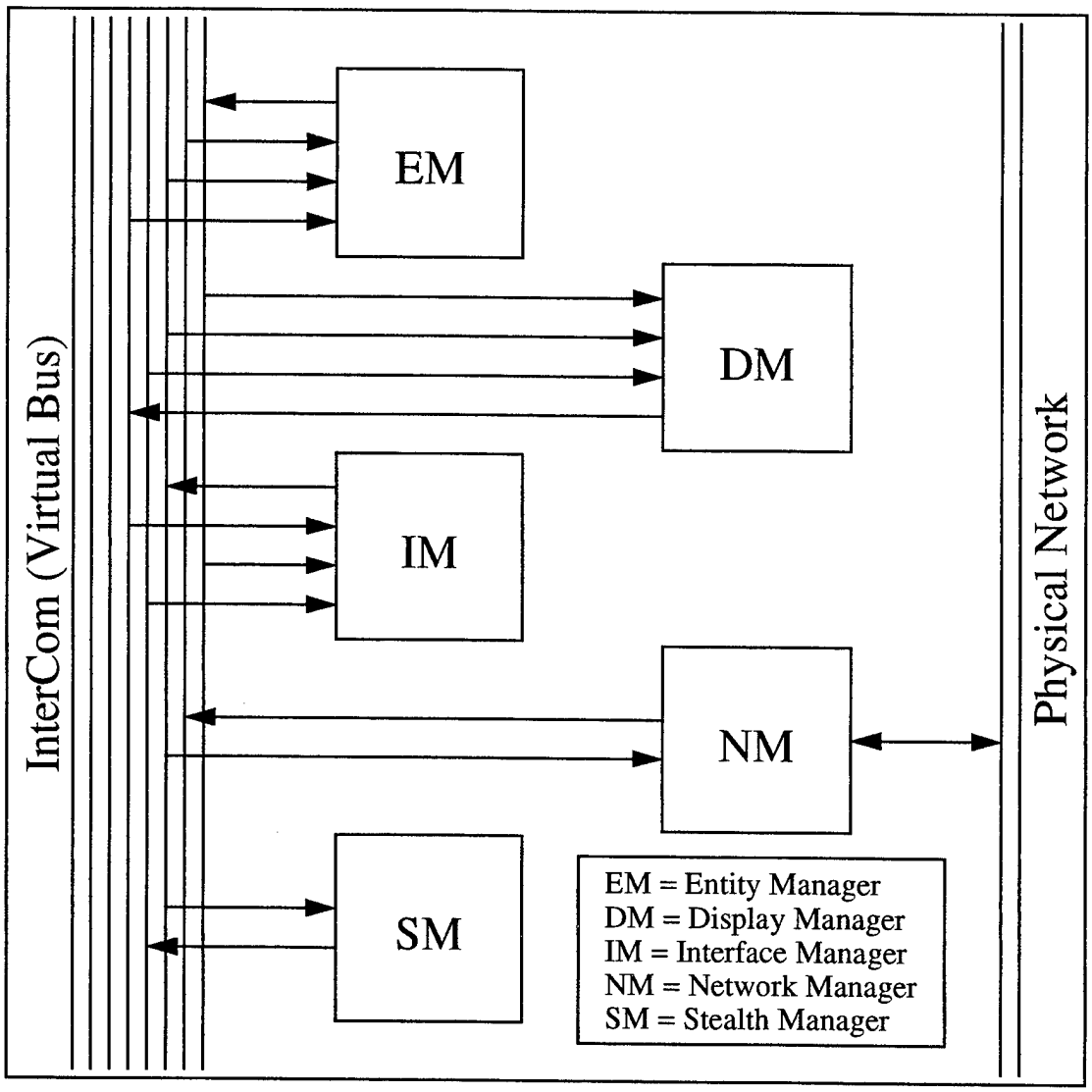


Figure 14. OpenNPNSET Architecture

a. Managers

A manager is a self-contained, independent thread of execution (lightweight thread, process, or program) with clearly defined responsibilities within the system. A

manager can use multiple code libraries and classes of objects to accomplish its mission. The libraries for OpenNPSNET are collections of related routines, with a well defined APIs, that provide a service. Typically, a library will make use of objects and maximize object-oriented principles such as modularity, reusability, data encapsulation and information hiding. Several libraries, with identical APIs, may exist to support cross-platform software development.

Each of the managers in the OpenNPSNET Architecture is designed to be a stand alone piece of software that can be run independently of the other managers. This modularity provides a great amount of flexibility for the system. Managers can be added or deleted from the system as the situation calls for, without affecting the other managers. In general, for a simulation system to run correctly it will require several managers to be running simultaneously. The Entity Manager is responsible for maintaining state information for all of the entities in the exercise. It will perform dead-reckoning, motion smoothing, and monitoring state and appearance changes. The Display Manager is responsible for generating and displaying the three-dimensional graphics representing the virtual world. It loads, maintains and displays the geometry for all entities and events in the world, manages the frame buffer and processes special display modes. The Interface Manager collects input data from all human-computer interfaces used in the simulation. It interprets the inputs and issues control information to accomplish the input requests. The Network Manager is responsible for all communications with other simulation systems within the exercise. It will read and process all information from the external network and publish that information for the other managers to use. It will also take information from the other managers and write it to the network in the specified message format. Other managers, such as the Stealth Manager, can be added as needed to manage other special purpose functions.

b. Intercom Library and Channels

The intercom library is used for all interprocess communications between managers. The library provides for multiple channels to support parallel communications. A channel is dedicated to one "type" or "class" of communication information. A manager may both publish and consume information from the same channel if needed. It is assumed that the intercom provides reliable communications. The API for this library will remain constant but its implementations will vary. At a minimum, the intercom library will include a TCP/IP Multicast version for use across a network and a shared memory version for use in a multi-processor computer.

c. Network Manager

The modular design of the OpenNPSNET Architecture provides a method for solving the protocol format dependency problem found with NPSNET-IV. By disassociating the internal data representation from the external communications protocol it is possible to implement a flexible protocol. The Network Manager serves as the API between the simulations system with its internal data format and the external simulation protocol.

Given the flexible nature of the external simulation protocol, the internal data representation must also be flexible. Because the simulation protocol may contain optional data units that may, or may not, be present in a PDU, the internal data representation must not depend on the format of the PDUs. Additionally, the data contained in a PDU may be in one of several presentations, so the internal data representation must be able to accommodate several possible formats for each data element. To accomplish this it is important that the internal formats used to communicate between managers on the intercom be flexible and capable of handling the largest data formats possible. In the OpenNPSNET Architecture, the method of communicating on the intercom is the Internal Protocol Data Unit (IPDU). The IPDU consists of a short header that identifies the contents of the IPDU and a variable amount of data. The header identifies the IPDU as either an Entity Attribute

IPDU or a Control IPDU. Entity Attribute IPDUs carry entity information and are identified by a unique Entity ID contained in the header. A Control IPDU carries control information used to run the simulation. The variable length data field contains data in a TAG : VALUE format. The TAG identifies the meaning and format of the data and the VALUE gives the actual data value. The names and formats of these TAGs and VALUEs is fixed and must be able of accommodating any of the possible presentations from the simulation protocol. As such, the worst case format is assumed. This should not be a problem, as these IPDUs will be transmitted using only shared memory or a Local Area Network, and network bandwidth constraints are not be a problem.

The Network Manger design is built around three software components. (See Figure 15) The first is a generic network reader/writer library that reads and writes PDUs to and from the network. This library is not dependent on the format of the PDUs and can read and write any PDU specified. Another software library reads and writes IPDUs to the intercom. This library is dependent on the TAGs and VALUE formats specified. There may

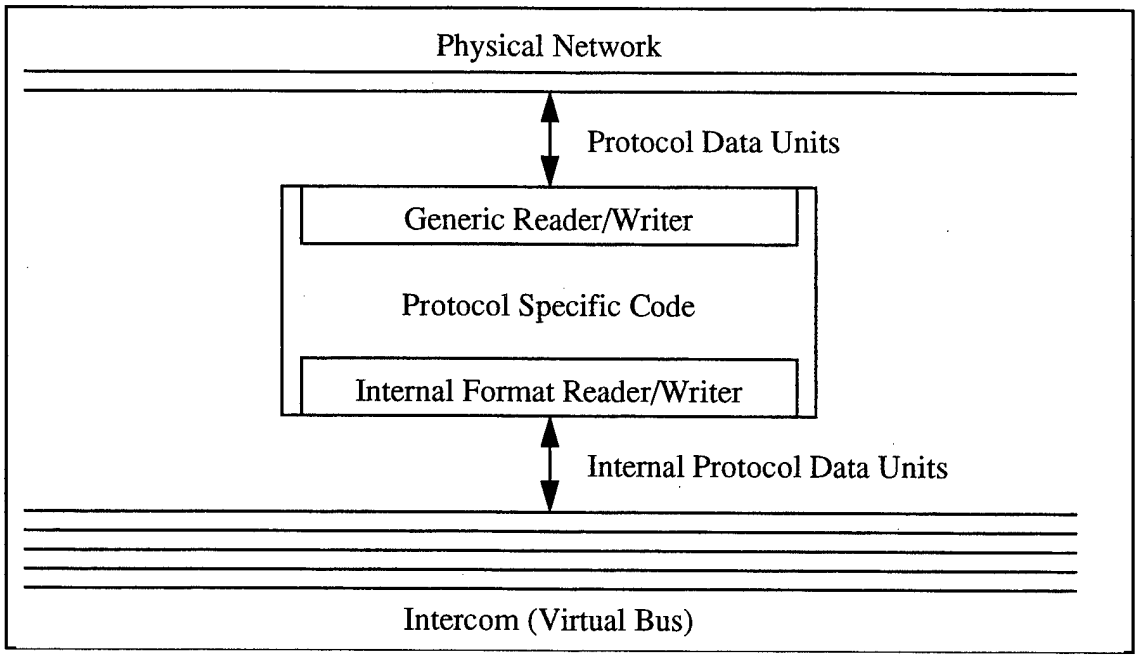


Figure 15. Network Manager Architecture

be several versions of each of these libraries for different platforms and for different implementations of the intercom.

The third software component is the simulation protocol specific code necessary to decode a PDU received from the network, map the data items received to the correct internal representation, and to format the IPDUs. This block of code will be highly dependent on the protocol specification and will be coded for each protocol.

4. Summary

The OpenNPSNET Architecture provides the application design that is necessary to utilize a reconfigurable simulation protocol. It allows the internal data representation to be disassociated from the external protocol format which, in turn, allows for a flexible protocol to be used. The Network Manager serves as the API between the external protocol and the internal data representation by decoding PDUs and mapping its data elements to IPDUs for internal use.

D. PROTOCOL SPECIFICATION GRAMMAR

1. Introduction

The benefit of a structured grammar lies in its ability to simplify the representation of a complex problem of concept. The Protocol Support Utility uses a modified Bachus-Naur Form grammar to describe the Protocol Data Units of the DIS Protocol. While the grammar used in the PSU is simple and it adequately describes the DIS Protocol, it provides for no flexibility within the PDUs it specifies. Once the protocol specification has been generated, it becomes fixed. The concept of Self-defined Messages with Multiple Presentations allows a protocol to be flexible and to optimize the content of PDUs at run-time, making more efficient use of network resources. Combining the formal grammar of the PSU with SDM/MP will create a formal method of generating a flexible protocol definition.

2. Grammar Definition

The grammar used by the Protocol Support Utility uses four distinct constructs for building a protocol. They are the Protocol Data Unit, the Composite Data Element, the Atomic Data Element, and the Primitive Data Type. While these four constructs are adequate for a fixed protocol definition, the SDM/MP concept requires a different set of constructs. The grammar used in this implementation also recognizes four constructs. They are the PDU, the Data Item, the Class, and the Primitive Data Types. Creating PDUs is the ultimate goal of this effort and PDUs are the highest level construct in the protocol. PDUs are constructed of Data Items. Each Data Item is an instance of a Class with one or more Presentations. A each Presentation of a Class is built of a Primitive Data Type. The Data Items in a PDU may be optional and may be present more than once.

The syntactic constructs used in this grammar are very similar to those used in the Protocol Support Utility. A PDU is identified by a pair of double angular brackets "<< >>". Data Items are distinguished by a pair of angular brackets, "<>", while Classes are represented by a character string. Optional Data Items are marked by either "TAG" or "FLAG" within braces "{TAG}". Arrays of Primitive Data Types or Data Items are indicated by square brackets "[3]". The complete list of syntactic constructs used is shown in Figure 16.

Using these constructs it is possible to build a protocol for almost any simulation requirement. It is possible to construct a simple, fixed protocol definition or a more complex, highly flexible protocol. Figure 17 gives an example of a very simple application of this grammar.

Grammar Construct	Meaning
<<PDU>>	Protocol Data Unit
<data item>	Data Item
Class	Class
enum8, uint16, ...	Primitive Data Type
{TAG}, {FLAG}	Optional Data Item
(P1)	Presentation
enum8[3]	Array of Primitive Data types
<data Item>[3]	Array of Data Items
::=	"is defined as"
;	"end definition"
+	Data Item may appear multiple times

Figure 16. Descriptive Grammar Constructs

As a naming convention, Class names begin with a capital letter while all other names begin with a lower case letter. The Primitive Data Types expressed in the grammar are, in reality, aliases for the basic data types found in the C++ programming language. Marking the existence of Optional Data Items with a flag is suitable when the data item has a high probability of appearing in the PDU and a small number of possible Presentations. Tags are included in the PDU using a bit-field. As a convention, the bit-field is located immediately after all mandatory data items. The bits in the bit-field are allocated in the order that the data items are listed in the PDU. The number of bits needed for each flagged, optional data items is found by rounding up the result of $\log_2(nP+1)$, where nP is the number of possible presentations for that data item [COHEN96b]. For example, a data item with only one presentation requires only one bit. A data item with two or three presentations requires two bits. The summation of these totals yields the total number of bits required in the bit-field. Cohen proposes that the format and size of the bit-field be calculated at setup time. He also proposes that the bit-field contain a "continuation" bit to indicate a continuation of the bit-field in another data item. While these concepts are certainly possible, for simplicity, they are not included in this implementation. It is up to

the protocol designer to calculate the bits required and to include a bit-field of the proper size in the PDU definition. It is also up to the protocol designer to determine if padding is needed to maintain alignment and to include it if necessary.

When reserving bits in the bit-field turns out to be expensive for data items that have a low probability inclusion or a large number of possible presentations, a tag can be used to indicate the presence of the data item. Again as a convention, all tags are grouped immediately following the bit-field for flagged data items. There is one tag for each optional data item included in the PDU. The size and format of the tag is dependent on the number of possible presentations for the data item. It is possible to represent 256 presentations of a data item with eight bits. This should normally be sufficient. The value of each tag will be calculated at setup time. Tag # 1 would be assigned to the first presentation of the first tagged, optional data item. Tag # 2 would go to the second presentation of the first tagged, optional data item etc. While this will add processing time at exercise setup, it should be minimal if tags are used correctly. For simplicity, it is again up to the protocol designer to determine the correct size and format for the tags and to include them in the PDU definition. Since 8 bits should normally be sufficient and all tags are grouped together, the tag would be included in the PDU as "<tag> (P1) +".

```
// Data Item Definitions

<pdu_type>                  ::=   Enumeration;
<time_stamp>                ::=   Time;
<sender_entityID>           ::=   Entity_ID;
<sender_linear_velocity>    ::=   Linear_Velocity;
<sender_location>           ::=   Location;
<tag>                       ::=   Tag;
<flag>                      ::=   Flag;

// Class Presentation Definitions

Enumeration       P1    ::=   enum8,
                  P2    ::=   enum16;

Time              P1    ::=   uint32,        // seconds
                  P2    ::=   uint16;        // mSec

Entity_ID         P1    ::=   uint16[3],     // site, host, entity #
                  P2    ::=   uint16,        // entity #
                  P3    ::=   uint16[2];     // site, host

Linear_Velocity   P1    ::=   float32[3],    // Vx, Vy, Vz [m/sec]
                  P2    ::=   float64[3],    // Vx, Vy, Vz [m/sec]
                  P3    ::=   int16[3];      // Vx, Vy, Vz [m/sec]

Location          P1    ::=   float64[3],    // X, Y, Z geo [m]
                  P2    ::=   int16[3];      // X, Y, Z geo [m]

Flag              P1    ::=   uint8,
                  P2    ::=   uint16,
                  P3    ::=   uint32;

Tag               P1    ::=   uint8,
                  P2    ::=   uint16;

<<EntityStatePDU>>      ::=   <pdu_type>              (P1),
                              <time_stamp>            (P1),
                              <sender_entityID>       (P2),
                              <flag>                  (P1),
                              <tag>                   (P1),
                              <sender_location>       (P1) (P2) {FLAG},
                              <sender_linear_velocity> (P1) (P2) {TAG};
```

Figure 17. Descriptive Grammar Example

3. Lexical Analyzer

Based on this grammar, a LEX-based scanner was developed. The scanner is a modification of the scanner used in the Protocol Support Utility and recognizes the unique delimiters which identify PDUs, Data Items, or Classes. As in the PSU, the scanner is designed as a simple finite state machine. The scanner states include:

<C_CMMT>

<CPP_CMMT>

<INITIAL>

<PDU_DEF>

<DATA_DEF>

<CLASS_DEF>

The states <C_CMMT> and <CPP_CMMT> support embedded comments in both the C and C++ programming language styles. In either of these states the scanner detects a comment and ignores all subsequent characters until it recognizes the appropriate comment-ending delimiter.

The <INITIAL> state is the reset state. In this state, the scanner is awaiting the definition of a protocol element. This state corresponds to the left-hand side of a production rule and the scanner will accept any valid grammar construct. In the <INITIAL> state, if a PDU, Data Item, or Class construct is detected, the <PDU_DEF>, <DATA_DEF>, or <CLASS_DEF> state is set as appropriate. In any of these states, the scanner expects to see only Data Items, Classes or Primitive Data Types. These three states correspond to the right-hand side of production rules.

When the scanner detects an allowable grammar construct, it will call one of the table management functions associated with the grammar. These functions initialize, load and update the internal data tables. The most important of these functions are listed in Figure 18.

Function	Purpose
void initTables();	Initializes the internal data tables.
void stripToken();	Removes the delimiting symbols.
void addSymbol();	Adds a symbol to the symbol table.
void addSymbolPDU();	Adds a PDU to the PDU table.
void addSymbolDATA();	Adds a Data Item to the tables.
void addSymbolCLASS();	Adds a CLASS to the CLASS table.
void addPRESENTATION();	Adds a Presentation to the tables.

Figure 18. Table Management Functions

Other functions are used to load array, optional data item, and other information into the tables. When the scanner reaches the end of the input grammar file, the internal data tables will be fully loaded. This process is used repeatedly during the development of a protocol specification as a method of initializing and updating the protocol specification stored in the internal data tables.

4. Internal Data Tables and Structures

The internal data tables are implemented as a series of fixed size array and serve as a repository for the data elements that are defined in a given grammar. These tables are accessed by both the lexical scanners and by the source code generation routines. Four separate tables are used to store the grammar constructs.

a. Symbol Table

The Symbol Table implementation is very similar to that used by Canterbury in the Protocol Support Utility. It contains the label, a character string that identifies each element of the protocol definition, and an index field that corresponds to one of the grammar constructs, PDU, Data Item, Class or Primitive Data Type. For each label stored in the Symbol Table, the appropriate index is used to match the label with the protocol

entity stored in its corresponding table. If the label corresponds to a PDU, then only the PDU index field is set. The others remain undefined. If the label corresponds to a Primitive Data Type, no index is defined. As the last step of the lexical analysis process, these Primitive Data Types are replaced with the appropriate C++ data types in the Data Item Table. In this way, the Symbol Table keeps track of the protocol element associated with each label.

b. PDU Table

The PDU Table contains the definitions of all of the PDUs contained in the protocol definition. Each entry in the PDU Table represents a single PDU definition. Each entry contains a label index that points at its name in the Symbol Table, an element count and an array of one or more data elements. The specific number of data elements in the PDU is contained in the element count.

c. Data Item Table

The Data Item Table matches the Data Item definition to a Class. The table contains a label index, pointing at the appropriate entry in the Symbol Table, and a Class Label index that points at the name of the appropriate class in the Symbol Table.

d. Class Table

The Class Table contains the definition of each class with all of the presentations for that class. The table consists of the label index, a presentation field indicating which presentation the entry represents, and an atomic type field. The atomic type field points at the entry in the Symbol Table for the Primitive Data Type used in the presentation. An additional field is used to indicate when an array of Primitive Data Types are used for that presentation.

E. SOURCE CODE GENERATION

The ability to define a protocol is of little value if there is no way to translate the protocol grammar into a usable form. One of the primary advantages of the Protocol

Support Utility is its ability to generate program source code based on the defined grammar. Because of the diversity of Distributed Simulation applications, the Protocol Support Utility adopted the concept of an Application Profile. An Application Profile is a collection of functions needed to produce the desired source code output for a given simulation system [CANTER95]. Application Profiles may differ greatly depending on the needs of the end system. Previously, the Protocol Support Utility supported two Application Profiles, the NPSNET Profile, used with NPSNET-IV and the Class-based Profile. As stated earlier, NPSNET-IV is not capable of effectively utilizing a flexible protocol, so the NPSNET Profile is of little value to this implementation. The Class-based Profile was designed to be an example of the Utility's ability to generate code based upon a different programming paradigm. While this Profile was not intended for any specific simulation, it formed the basis of the development of an Application Profile for the first simulation implemented using the OpenNPSNET Architecture, NPSNET-V.

The initial implementation uses a combination of the Protocol Support Utility's Class-based Profile and the current DIS Network Class. The source code generated by the PSU implements each PDU as a class. Each of the PDU classes contains the data items that make up the PDU and the functions necessary to set the data item values of the PDU, calculate the length of the PDU, create the bit-mask and to write the PDU to the network. To determine which optional data items should be included in the PDU at which presentation, there is a 'present' flag that is set to 1 if the optional data item is to be transmitted or to 0 if the data item is not present. This flag is set by the set value function when the application builds the PDU. The member functions include write_PDU(), read_PDU(), reset_PDU(), print_PDU(), pdu_length(). If there are optional data items, functions set_bit_field() and set_present() will be generated. For initial testing, the write_PDU() and read_PDU() functions are implemented using the existing DIS Network Library. This was done to simulate the OpenNPSNET Network Manager which is under development. An example of this source code is included in Appendix C.

F. SUMMARY

The combination of the formal, grammar based protocol definition used in the Protocol Support Utility and the concepts of Self-Defined Messages and Multiple Presentations, as proposed by Cohen, has resulted in a more flexible method of defining a protocol for use by distributed simulations. Such a protocol is easily optimized at run time so that the network resources used can be minimized. This occurs by eliminating all unnecessary information contained in a PDU through the use of optional data items, and, by utilizing the smallest number of bits to represent the information that is contained in the PDU through the concept of Multiple Presentations. By minimizing the network resources necessary for a simulation application to communicate, it will be possible to increase the number of simulation entities participating in the exercise. This method also increases the extensibility of the protocol by allowing the user to define any type of PDU necessary.

VI. EXPERIMENTAL RESULTS

A. SOURCE CODE TESTING

To test the generation of source code by the Protocol Support Utility, two current DIS utility programs were modified to use the generated class libraries. POPULATE is a DIS compliant program that is used to create and transmit DIS PDUs to a network. It was initially developed as a simple method of sending PDUs across a network as an aid to simulation development. POPULATE uses pre-programmed state information to generate Entity State PDUs. BUG_DIS reads Entity State PDUs from the network, copies the data from the received PDU into a new PDU and sends it back out onto the network. It was developed as an irritant program for harassing entities in a DIS exercise. These two applications were chosen for this test because of their simplicity and focus on the Entity State PDU. Both were modified to use the generated classes and to provide for network monitoring.

The testing focused on the Entity State PDU because of its dominance of network traffic during a simulation exercise. Because approximately 70% of all network traffic is ESPDUs, the greatest gains in network utilization should be realized by optimizing the ESPDU. The base case for this test consists of entities transmitting the current DIS 2.0.3 ESPDU while in a resting state. It is in the resting state when the largest number of unnecessary bits are transmitted across the network. Using the PSU, a protocol specification of the current DIS ESPDU, with all data items mandatory and with the correct DIS 2.0.3 presentations, was created (See Figure 19) and source code for the network classes generated. The test consisted of POPULATE creating and transmitting ESPDUs across the network while it maintained a log of the information that it had sent. BUG_DIS received the PDUs and retransmitted the data as it kept a record of the PDUs received. This established the base measurement for comparison to the test case.

```
<<EntityStatePDU>>  ::=    <protocol_version>(P1),
                           <exercise_ID>(P1),
                           <PDU_Type>(P1),
                           <protocol_family>(P1),
                           <time_stamp>(P1),
                           <length>(P2),
                           <header_padding>(P2),
                           <sender_entityID>(P1),
                           <force_ID>(P1),
                           <num_articulation_params>(P1),
                           <entity_kind>(P1),
                           <domain>(P1),
                           <country>(P2),
                           <category>(P1),
                           <sub_category>(P1),
                           <specific>(P1),
                           <extra>(P1),
                           <alt_entity_kind>(P1),
                           <alt_domain>(P1),
                           <alt_country>(P2),
                           <alt_category>(P1),
                           <alt_sub_category>(P1),
                           <alt_specific>(P1),
                           <alt_extra>(P1),
                           <sender_linear_velocity>(P1),
                           <sender_location>(P1),
                           <sender_orientation>(P1),
                           <sender_appearance>(P3),
                           <DeadReckAlgorithm>(P1),
                           <OtherParams>(P4),
                           <sender_linear_accel>(P1),
                           <sender_angular_vel>(P1),
                           <character_set>(P1),
                           <markings>(P5),
                           <capabilities>(P3);
```

Figure 19. Specification of Base Case ESPDU

The test case consisted of an ESPDU with all but the critical information made optional. For this test the critical data items were determined to be all of the PDU header information, the entity kind information, sender location and sender orientation. These were determined to be the minimum information necessary to correctly relay the state of an entity at rest. Data items such as location and orientation were given presentations that

required fewer bits than the DIS 2.0.3 presentations. All optional data items were marked with a FLAG and were given only one presentation. This method was chosen because it resulted in the fewest number of bits to mark the presence of the optional data items.The specification for this ESPDUs shown in Figure 20.

```
<<EntityStatePDU>>  ::=    <protocol_version>(P1),
                           <exercise_ID>(P1),
                           <PDU_Type>(P1),
                           <protocol_family>(P1),
                           <time_stamp>(P1),
                           <length>(P2),
                           <header_padding>(P2),
                           <sender_entityID>(P1),
                           <force_ID>(P1),
                           <num_articulation_params>(P1),
                           <entity_kind>(P1),
                           <domain>(P1),
                           <country>(P2),
                           <category>(P1),
                           <sub_category>(P1),
                           <specific>(P1),
                           <extra>(P1),
                           <sender_location>(P3),
                           <sender_orientation>(P2),
                           <flag>(P2),
                           <alt_entity_kind>(P1) {FLAG},
                           <alt_domain>(P1) {FLAG},
                           <alt_country>(P2) {FLAG},
                           <alt_category>(P1) {FLAG},
                           <alt_sub_category>(P1) {FLAG},
                           <alt_specific>(P1) {FLAG},
                           <alt_extra>(P1) {FLAG},
                           <sender_linear_velocity>(P1) {FLAG},
                           <sender_appearance>(P3) {FLAG},
                           <DeadReckAlgorithm>(P1) {FLAG},
                           <OtherParams>(P4) {FLAG},
                           <sender_linear_accel>(P1) {FLAG},
                           <sender_angular_vel>(P1) {FLAG},
                           <character_set>(P1) {FLAG},
                           <markings>(P5) {FLAG},
                           <capabilities>(P3) {FLAG};
```

Figure 20. Specification of Test Case ESPDU

The first run of the test case consisted of sending ESPDUs for a stationary entity with the required data items, header information, entity type, and location, at the smallest

possible presentations. This configuration establishes the greatest possible reduction in bits transmitted while using the data items from the base case. Other configurations of the test case were tried with the entities moving and using larger presentations of the data items. It was expected that the test results from these configurations would lie somewhere between the base case and the first test case.

B. TEST RESULTS

The testing demonstrated that both the base case protocol specification and the test case protocol specification could be used to generate source code that could read and write ESPDUs from the network. The test case demonstrated that by using optional data items it is possible to significantly reduce the number of bits transmitted across the network. While the base case ESPDU required 144 bytes to be transmitted when the entity was at rest, the test case ESPDU required only 42 bytes. This is a reduction of almost four times.

This reduction in the number of bytes transmitted can have a significant impact for a large-scale simulation. Assuming the current DIS Standard of at least one ESPDU per entity every five seconds, a simulation of 100,000 entities with each entity at rest would require approximately 23 Mega-bits per second (Mbps) of network bandwidth when using the current DIS ESPDU. Using the ESPDU defined in Figure 20, the network bandwidth required drops to approximately 5.9 Mbps, a decrease of 3.9 times. Of course, this is the maximum possible reduction. In reality, a large percentage of entities would be changing state by moving or interacting with other entities by firing, colliding or some other manner. The actual reduction in network bandwidth requirements would be somewhere between the two extremes. The specific requirements are impossible to predict because of the highly dynamic nature of network traffic during a distributed simulation.

The techniques of Self-Defined Messages and Multiple Presentations provide a mechanism to significantly reduce the network bandwidth requirements of a large-scale distributed simulation. These reductions will make the realization of 100,000 player simulations more likely.

VII. FINDINGS AND FUTURE RESEARCH

The future of DIS will largely depend on its ability to adapt to the growing user demand for distributed simulations and to support the larger, more dynamic virtual worlds of tomorrow. The "scalability" and "extensibility" of the protocol are the principle issues that must be addressed in order to meet these new demands. Scalability requires that the protocol be able to perform adequately as the number of entities in the simulation grows. Extensibility requires the protocol to be easily modified to meet new simulation demands. The current DIS Protocol is neither scalable nor extensible. This thesis has addressed these problems by developing a method of easily creating or modifying a protocol that can be optimized during execution. The Protocol Support Utility Version 2, provides a means of easily creating a protocol specification using a simple BNF style grammar. The grammar used supports the scalability of the protocol through optional data items and multiple presentations. Through these concepts it is possible to minimize the number of bits transmitted across the network by sending only the data necessary to convey the state of the simulation and by sending that data in the smallest number of bits that will meet the needs of the simulation.

This chapter provides a summary of the findings and conclusions resulting from this research. The following information reflects the insight gained in formulating a descriptive grammar for simulation protocols, modifying the Protocol Support Utility to use the new grammar, and in the application of these tools. Also presented is a synopsis of related research topics which require continued investigation.

A. FINDINGS

This thesis effort was organized into four distinct phases. The results of this work are similarly organized. The first phase was to design a simulation protocol that would be both scalable and extensible. To meet this goal the concepts of Self-Defined Messages and Multiple Presentations were incorporated with the current DIS Protocol. The Self-Defined Message concept allows for data items within the PDUs to be optional and for the

simulation application to decide at run-time which of these optional data items should be included in the PDU. Multiple Presentations allows data items to be represented in several ways and for the application to decide at run-time the most appropriate representation of the data. These concepts create a protocol that is easily optimized at run-time and uses the minimum number of bits required to transmit the state of the simulation.

The second phase was to develop a formal grammar to describe the protocol discussed above. A modified Bachus-Naur Form grammar was successfully used to describe the protocol. This grammar provides a simple method of rapidly defining a simulation protocol. The grammar developed is able to accurately define a simulation protocol for any use.

The third phase was to adapt the Protocol Support Utility to use the new grammar and to generate source code for reading and writing the new protocol. The Protocol Support Utility was easily modified to support the new protocol specification. Source code generation proved to be troublesome because of the lack of a clearly defined API between the network protocol and the application. A more generic network library that is not dependent on specific PDU definitions is needed to serve as this API. The Network Manager of the OpenNPSNET Architecture will provide this functionality.

The final phase of this research was the testing of the protocol. A simple test of the implementation of the software necessary to use this protocol demonstrated that a maximum of a 400% decrease in the number of bits transmitted across the network can be realized. The network traffic of a distributed simulation is usually very dynamic and this figure represents the best possible optimization of this traffic. In practice, the reduction in network traffic would be somewhat less than this. These results show that the new protocol can significantly reduce the network resources required and will allow for a greater number of participants in a large-scale distributed simulation.

B. AREAS FOR FUTURE RESEARCH

Distributed Simulations are an evolving technology and future research opportunities can be found in many areas. In [DIS94] , the DIS community has defined its objectives for future research. Many of these objectives are yet to be met and much work remains to be done. There are a number of topics which stem specifically from this work.

First, this effort should be continued by fully implementing the Network Manager of the OpenNPSNET Architecture and fully testing the SDM/MP Protocol. A complete understanding of the benefits and drawbacks of this approach will not be realized until testing with a full simulation application is done. Additional testing of different protocol specifications must also be completed.

Future research should examine the possibility of removing the code generation step from the process. It may be possible for a simulation application to directly access the tables used by the Protocol Support Utility to determine the protocol specification. It seems possible that a generic PDU reader/writer could be developed that uses the protocol specification stored in the symbol tables to process PDUs. This would simplify the protocol specification process and remove the time consuming step of recompiling the network libraries every time the protocol specification is changed.

Another area related to this effort that requires research is the distribution of the protocol specification. For this approach to work well for a large-scale distributed simulation, there must be a method of easily and rapidly distributing a new protocol specification to all participants in the simulation. Ideally, this method would be transparent to the participants and could be done while the simulation is running. Possible approaches include transmitting the protocol specification in a special PDU or as an autonomous agent using an agent system such as Telescript or Java.

Addition of a compliance validation feature to the Protocol Support Utility would be of great benefit. This feature would provide a means to verify that a new or modified protocol specification is consistent with a given protocol standard. Addition of this feature

would extend the utility of the tool to include Protocol Verification, Validation and Accreditation (VV&A) as well as ongoing Configuration Management Efforts.

While this research has focused on optimizing the amount of network traffic during a distributed simulation, it has not addressed the problem of each entity having to process every PDU transmitted. In fact, this effort may complicate this problem because of the additional processing required to create and decode the optional data items in PDUs. The Area of Interest Manager proposed in [MAC95a] must be implemented in order to solve this problem. This will require a great amount of work and development should begin soon.

APPENDIX A. PROTOCOL SUPPORT UTILITY INSTRUCTIONS

DIS PROTOCOL SUPPORT UTILITY Version 2 USER INFORMATION

The Protocol Support Utility is intended as a tool to be used in the development and refinement of DIS data elements. The following describes the installation, operation, use, and support of this tool.

System Hardware Requirements.....

The Protocol Support Utility was developed for use on platforms which support typical UNIX-hosted, X-Window environments. Beyond the need for minimal runtime disk space, no other special resources or hardware facilities are required.

Installation.....

The following files are required for proper program operation. All files should be installed in the same directory in which the Utility is to be used.

File	Use
psu	Protocol Support Utility
_Sample.gram	Default/initialization grammar

The following files are required to support source code generation for the Application Profiles specified:

Class-based Profile

enum.h Enumeration data file (source:NPSNET pdu.h)

Silicon Graphics provides an ASCII-based text editor, jot, for default use on most SGI workstations. The Protocol Support Utility uses this program as the principle means of editing the protocol grammar. To ensure proper operation of this feature, jot should be installed in the following location:

/usr/sbin/jot

This is the default installation path for jot on most systems. If jot is not available, or another editor is preferred, an alternate program may be used. However, the use of an alternate editor requires that minor changes be made to the Protocol Support Utility, and the entire system recompiled. The specific source code changes required are discussed below.

Program Initiation.....

The Protocol Support Utility may be initiated from the UNIX command line as follows:

% psu

If desired, an initial grammar file may be loaded on program start-up by providing the name of the desired grammar file when invoking the Utility as follows:

% psu myGrammar.gram

User Interface.....

The Protocol Support Utility consists of a Main Program Window which supports a pulldown menu bar, a display window for runtime status messages, and a viewing window in which working grammar definitions may be displayed. Also provided are buttons to initiate the editing and update of specific grammar constructs.

The pulldown menu bar has two options, File and Generate. The File menu is used to manage the grammar files processed by the Utility, while the Generate option is used to initiate source code production as appropriate. The particular facilities provided under each option are discussed below.

The Grammar View window provides a facility to view the grammar-based representation of DIS protocol elements. This is window is for display only and editing is not supported. The contents of the viewing window may be selected by means of the pulldown widget labeled "Grammar View". This widget is located directly above the viewing window and offers the following selections:

--Sample - allows viewing of a sample grammar specification.
--PDU - allows viewing of all PDUs defined in the current working file.
--Class - allows viewing of all Classes defined.
--Data Item - allows viewing of all Data Items defined.
--All - allows viewing of all PDUs, Classes, and Data Items defined.

The Grammar View window may be scrolled to view protocol definitions which have been selected for viewing but do not appear within the text window. If necessary, the window may be refreshed by reselecting the particular grammar view desired.The Runtime Message window is located in the lower portion of the Main Window. This purpose of this window is to display the

program's runtime status and error messages as they occur. Like the Grammar View display, this window may be scrolled to view previous messages which do not appear within the current viewing area.Located immediately above the Runtime Message window is the Working Grammar File window. This single line, text window displays the path and filename of the grammar file currently active in the Utility. Like the Grammar View and Runtime Message windows, the Working Grammar File window is used for the display of information only and may not be edited.

Grammar Files.....

The File option on the Menu Bar provides facilities to Open and Merge pre-existing grammar files, as well as created New files for protocol development work. It should be noted that any grammar file loaded must be in a format consistent with the DIS descriptive grammar specified for use with this Utility. A brief discussion of the grammar is presented later in this guide.Also found under the File option are facilities to Save the working grammar file, Close the working grammar file, and Exit the program.

Authoring and Editing.....

This Utility supports the use of a simple grammar to describe DIS protocol entities. The Grammar has three basic constructs:

<< ProtocolDataUnits >>
< Data Items >
Classes

These constructs are used to model the individual protocol entities associated with DIS. The following example illustrates the use of this grammar in describing a typical DIS entity:

```
Enumeration P1:: =      enum8,
                P2::=    enum16;


<protocol_version>::=   Enumeration;

<<SamplePDU>>::=        <protocol-version>(P1),
                        <PDU_type>(P2);
```

Note that Class definitions should be listed first, followed by data item definitions and then PDU definitions. If definitions are not in this order, the output of the Protocol

Support Utility is unpredictable. For a more detailed explanation of this protocol definition method consult "A Rapidly Reconfigurable, Application Layer, Virtual Environment Network Protocol".

Naming conventions are an important consideration when describing protocol entities. Care should be taken that the names given to protocol entities be consistent with those used in the system for which code is to be generated. Failure to do so will result in compilation problems when the generated code is integrated into the DIS host system.

DIS descriptive grammars may be authored or edited using any ASCII-based text editor. As mentioned above, the editing facility incorporated into the Protocol Support Utility is jot, a SGI-supplied product. Grammar edit functions are initiated by depressing the Edit button associated with the particular DIS protocol construct to be edited. When a particular edit function is invoked, a jot session is launched. When invoked, the jot editor will contain the grammar description of the DIS entities associated with the current session.
For example, the PDU Editor may be launched by depressing the Protocol Data Unit Edit button. This done, a jot session will appear and the session window will contain all PDUs currently defined in the Utility's internal PDU table.

The grammar displayed in any jot session may be edited as desired. The filename reflected by the jot editor (seen above the jot window) is a scratch file created by the Utility. Once editing is complete, this file must be saved if changes to the grammar are to be made.Once the altered grammar file has been saved, the jot editor may be exited. To effect any changes made during an edit session, the corresponding Update button must be depressed. Initiating the Update function will effectively reinitialize the internal symbol tables within the Utility, thereby incorporating any changes made to the grammar. There is no "undo" feature currently implemented in the Utility.

Source Code Generation.....

Code Generation is an automated process. Any working grammar may be used to generate DIS program source code. A working grammar is the grammar definition in use by the program at the time that Code Generation is initiated.

As mentioned earlier, the file associated with the working grammar is displayed in the Working Grammar File window.To initiate Code Generation, select the appropriate Applications Profile under the Generate menu located on the Main Menu Bar. Currently, only one Applications Profile has been implemented. The Class-based Profile, is intended for use with NPSNET version 5. This profile is not

complete as the final configuration of NPSNET-V is not complete.

Additional Profiles may be added to the Utility if desired. Similarly, the current Profile may be modified or improved if required.

Program Termination.....

The Protocol Support Utility may be terminated by selecting the Exit option in the File menu located on the Main Menu Bar or by closing ("double-clicking") the main program window. Care should be taken to save all work prior to program termination.

APPENDIX B. PROTOCOL GRAMMAR SOURCE CODE

```
Program:    DIS LEX

Date:       6 May 1996

Written by: Steve Stone

Purpose:    This program is part of a larger system
            used to parse a user prepared grammar. The
            grammar used is a modified version of BNF
            and is used to describe the application
            protocol applicable to Distributed Inter-
            active Simulations (DIS). This grammar
            supports protocols using the concepts of
            Self-defined Messages and Multiple Presentations.

%s C_CMMT C_CMMT_PDU C_CMMT_DATA C_CMMT_CLASS
%s CPP_CMMT CPP_CMMT_PDU CPP_CMMT_DATA CPP_CMMT_CLASS
%s PDU_DEF DATA_DEF CLASS_DEF

%{


#include <string.h>
#include <stdlib.h>

#include "disPSU_GLOBAL_DEFINES.h"
/* #include "genCLASS_globals.h" - only atomicCOUNT referenced */
#include "genCLASS_funcs.h"


char tokenString[MAX_STRING];
char structString[MAX_STRING];
char dataString[MAX_STRING];

extern int classCOUNT = 0;

int  LHS = 1;
int  ix;

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*   The below function strips the special delimiter symbols     */
/*  from the scanned tokens.                                     */

void stripToken(char* inString,int tokenType, int yyleng)
{
  int  count;
  char *tempString = "";
```

```
  for(count=tokenType;count<(yyleng-tokenType);count++)
    {
    if (inString[count] != ' ')
      tempString[count-tokenType] = inString[count];
    }
  tempString[count-tokenType] = '\0';
  strcpy(tokenString,tempString);
}


%}

%%

[ \t]*                                  { /*  skip whitespace  */ };
<INITIAL>[ \n]*          |
<PDU_DEF>[ \n]*          |
<DATA_DEF>[ \n]*|
<CLASS_DEF>[ \n]*        { /*  end of line  */ };

<INITIAL>"/*"            { BEGIN C_CMMT; };
<PDU_DEF>"/*"            { BEGIN C_CMMT_PDU; };
<DATA_DEF>"/*"           { BEGIN C_CMMT_DATA; };
<CLASS_DEF>"/*"          { BEGIN C_CMMT_CLASS; };

<C_CMMT>[^*/]          ;
<C_CMMT_PDU>[^*/]      ;
<C_CMMT_DATA>[^*/]     ;
<C_CMMT_CLASS>[^*/]    ;

<C_CMMT>"*/"             { BEGIN INITIAL; };
<C_CMMT_PDU>"*/"         { BEGIN PDU_DEF; };
<C_CMMT_DATA>"*/"        { BEGIN DATA_DEF; };
<C_CMMT_CLASS>"*/"       { BEGIN CLASS_DEF; };

<INITIAL>"//"            { BEGIN CPP_CMMT; };
<PDU_DEF>"//"            { BEGIN CPP_CMMT_PDU; };
<DATA_DEF>"//"           { BEGIN CPP_CMMT_DATA; };
<CLASS_DEF>"//"          { BEGIN CPP_CMMT_CLASS; };

<CPP_CMMT>[^\n]*         ;
<CPP_CMMT_PDU>[^\n]*     ;
<CPP_CMMT_DATA>[^\n]*    ;
<CPP_CMMT_CLASS>[^\n]*   ;

<CPP_CMMT>[\n]           { BEGIN INITIAL; };
<CPP_CMMT_PDU>[\n]       { BEGIN PDU_DEF; };
<CPP_CMMT_DATA>[\n]      { BEGIN DATA_DEF; };
<CPP_CMMT_CLASS>[\n]     { BEGIN CLASS_DEF; };
```

```
<INITIAL>\<\<[A-Za-z_][-A-Za-z0-9_.]*\>\>

                        { BEGIN PDU_DEF;
                          stripToken(yytext,2,yyleng);
                          addSymbolPDU(tokenString);
                        };

<INITIAL>\<[A-Za-z_][-A-Za-z0-9_.]*\>

                        { BEGIN DATA_DEF;
                          stripToken(yytext,1,yyleng);
                          addSymbolDATA(tokenString, LEFT, NONE);
                        };

<INITIAL>[A-Za-z_][-A-Za-z0-9_.]*

                        { BEGIN CLASS_DEF;
                          stripToken(yytext,0,yyleng);
                          addSymbolCLASS(tokenString, LEFT, NONE);
                        };

\:\:\=                  {LHS = 0; };


<CLASS_DEF>[-A-Za-z0-9_.]*

                        { stripToken(yytext,0,yyleng);
                          if (LHS)
                          {
                            addPRESENTATION(tokenString);
                          }
                          else
                          {
                            addSymbolATOMIC(tokenString);
                          };
                        };

<PDU_DEF>\<[A-Za-z_][-A-Za-z0-9_.]*\>

                        { stripToken(yytext,1,yyleng);
                          addSymbolDATA(tokenString, RIGHT, NONE);
                        };

<PDU_DEF>\([0-9_.]*\)

                        { stripToken(yytext,1,yyleng);
                          addPDUPresentation(tokenString);
                        };


<PDU_DEF>\{[A-Za-z_][-A-Za-z0-9_.]*\}
```

```
                        { stripToken(yytext,1,yyleng);
                          setOptionFlag(tokenString);
                        };

<PDU_DEF>\[[A-Za-z_][-A-Za-z0-9_.]*\]

                        { stripToken(yytext,1,yyleng);
                          addPDU_Array(tokenString);
                        };

<PDU_DEF>\+
                        { stripToken(yytext,1,yyleng);
                          setMultiFlag();
                        };

<CLASS_DEF>\[[-A-Za-z0-9_*+/.]*\]

                        { stripToken(yytext,1,yyleng);
                          addCLASS_Array(tokenString);
                        };

<DATA_DEF>[-A-Za-z0-9_.]*

                        { stripToken(yytext,0,yyleng);
                          addSymbolCLASS(tokenString, RIGHT, NONE);
                        };


<CLASS_DEF>","          { LHS = 1; };

<PDU_DEF>","            { reset_pduPresentation(); };

<INITIAL>\;             { LHS = 1;
                          objectCOMPLETE(CLASS_DEF_STATE);
                        };

<PDU_DEF>\;             { BEGIN INITIAL;
                          LHS = 1;
                          objectCOMPLETE(PDU_DEF_STATE);
                        };

<DATA_DEF>\;            { BEGIN INITIAL;
                          LHS = 1;
                          objectCOMPLETE(DATA_DEF_STATE);
                        };



<CLASS_DEF>\;           { BEGIN INITIAL;
```

```
                    LHS = 1;
                    objectCOMPLETE(CLASS_DEF_STATE);
                  };

.                   ;
%%

int yywrap()
  {
  fclose(yyin);
  return 1;
  }
```

APPENDIX C. GENERATED SOURCE CODE

```
//************************************************************
//                    Protocol Support Utility
//                          Version 2
//
//               Automated Source Code Generation
//                          for use with
//
//                CLASS-based DIS Implementations
//
//************************************************************


#ifndef __CLASS_PDU__
#define __CLASS_PDU__

#undef  DIS_1_0
#define DIS_2_0

#include "disnetlib.h" // For testing only!
#include "pdu.h" // For testing only!
#include <iostream.h> // For testing only!

/*************************************************************/
/****                      Class-based                      ****/
/****          DIS Protocol Data Unit Definitions           ****/
/*************************************************************/

/* Official PDU Types: Required */

#define entity_state_PDU_Type                0

class entity_state_PDU
{
       public:
             entity_state_PDU();
             ~entity_state_PDU();
             int  write_PDU();
             int  read_PDU();
             void  reset_PDU();
             void  print_PDU();
             int  pdu_length();
             void  set_bit_field();
             void  set_present();
             void  display_bits(unsigned short value);
```

```
//*****************************For Testing Only*************************
            int entity_state_PDU::operator==(const entity_state_PDU &);
            int entity_state_PDU::operator!=(const entity_state_PDU &);
//*****************************For Testing Only*************************

        unsigned char               protocol_version;
        unsigned char               exercise_ID;
        unsigned char               PDU_Type;
        unsigned char               protocol_family;
        unsigned int                time_stamp;
        unsigned short              length;
        short                       header_padding;
        unsigned short              sender_entityID[3];
        unsigned char               sender_force_ID;
        unsigned char               num_articulation_params;
        unsigned char               entity_kind;
        unsigned char               domain;
        unsigned short              country;
        unsigned char               category;
        unsigned char               sub_category;
        unsigned char               specific;
        unsigned char               extra;
        short                       sender_location[3];
        short                       sender_orientation[3];
        unsigned short              flag;
        unsigned char               alt_entity_kind;
        int                         alt_entity_kind_PRESENT;
        unsigned char               alt_domain;
        int                         alt_domain_PRESENT;
        unsigned short              alt_country;
        int                         alt_country_PRESENT;
        unsigned char               alt_category;
        int                         alt_category_PRESENT;
        unsigned char               alt_sub_category;
        int                         alt_sub_category_PRESENT;
        unsigned char               alt_specific;
        int                         alt_specific_PRESENT;
        unsigned char               alt_extra;
        int                         alt_extra_PRESENT;
        float                       sender_linear_velocity[3];
        int                         sender_linear_velocity_PRESENT;
        unsigned int                sender_appearance;
        int                         sender_appearance_PRESENT;
        unsigned char               DeadReckAlgorithm;
        int                         DeadReckAlgorithm_PRESENT;
        unsigned char               OtherParams[15];
        int                         OtherParams_PRESENT;
        float                       sender_linear_accel[3];
        int                         sender_linear_accel_PRESENT;
        float                       sender_angular_vel[3];
        int                         sender_angular_vel_PRESENT;
```

```
        unsigned char                   character_set;
        int                             character_set_PRESENT;
        unsigned char                   markings[11];
        int                             markings_PRESENT;
        unsigned int                    capabilities;
        int                             capabilities_PRESENT;

//**********************for testing only***************
   DIS_net_manager *net;
//**********************for testing only***************

      private:

};

// Function: <<
// Purpose:  overloaded ostream operator - expects an object as its
argument

inline ostream & operator<<(ostream & output, entity_state_PDU & e)
           {e.print_PDU();  return output;}


// Function: <<
// Purpose:  overloaded ostream operator - expects a pointer to an
object as
//          its argument

inline ostream & operator<<(ostream & output, entity_state_PDU *e)
           {e->print_PDU();  return output;}

#endif      /* DIS_1_0 */
#endif  /* _CLASS_PDU  */
```

```
//************************************************************
//                    Protocol Support Utility
//                          Version 2
//
//               Automated Source Code Generation
//                         for use with
//
//                CLASS-based DIS Implementations
//
//************************************************************


#include "pduCLASS.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//   Member Functions : entity_state_PDU

entity_state_PDU::entity_state_PDU()
{
   reset_PDU();
}

entity_state_PDU::~entity_state_PDU()
{   }

void entity_state_PDU::reset_PDU()
{
   protocol_version                    = '\0';
   exercise_ID                         = '\0';
   PDU_Type                            = '\0';
   protocol_family                     = '\0';
   time_stamp                          =    0;
   length                              =    0;
   header_padding                      =    0;
   sender_entityID[0]                  =    0;
   sender_entityID[1]                  =    0;
   sender_entityID[2]                  =    0;
   sender_force_ID                     = '\0';
   num_articulation_params             = '\0';
   entity_kind                         = '\0';
   domain                              = '\0';
   country                             =    0;
   category                            = '\0';
   sub_category                        = '\0';
   specific                            = '\0';
   extra                               = '\0';
   sender_location[0]                  =    0;
```

```
sender_location[1]                    =    0;
sender_location[2]                    =    0;
sender_orientation[0]                 =    0;
sender_orientation[1]                 =    0;
sender_orientation[2]                 =    0;
flag                                  =    0;
alt_entity_kind                       = '\0';
alt_entity_kind_PRESENT               =    0;
alt_domain                            = '\0';
alt_domain_PRESENT                    =    0;
alt_country                           =    0;
alt_country_PRESENT                   =    0;
alt_category                          = '\0';
alt_category_PRESENT                  =    0;
alt_sub_category                      = '\0';
alt_sub_category_PRESENT              =    0;
alt_specific                          = '\0';
alt_specific_PRESENT                  =    0;
alt_extra                             = '\0';
alt_extra_PRESENT                     =    0;
sender_linear_velocity[0]             =  0.0;
sender_linear_velocity[1]             =  0.0;
sender_linear_velocity[2]             =  0.0;
sender_linear_velocity_PRESENT        =    0;
sender_appearance                     =    0;
sender_appearance_PRESENT             =    0;
DeadReckAlgorithm                     = '\0';
DeadReckAlgorithm_PRESENT             =    0;
OtherParams[0]                        = '\0';
OtherParams[1]                        = '\0';
OtherParams[2]                        = '\0';
OtherParams[3]                        = '\0';
OtherParams[4]                        = '\0';
OtherParams[5]                        = '\0';
OtherParams[6]                        = '\0';
OtherParams[7]                        = '\0';
OtherParams[8]                        = '\0';
OtherParams[9]                        = '\0';
OtherParams[10]                       = '\0';
OtherParams[11]                       = '\0';
OtherParams[12]                       = '\0';
OtherParams[13]                       = '\0';
OtherParams[14]                       = '\0';
OtherParams_PRESENT                   =    0;
sender_linear_accel[0]                =  0.0;
sender_linear_accel[1]                =  0.0;
sender_linear_accel[2]                =  0.0;
sender_linear_accel_PRESENT           =    0;
sender_angular_vel[0]                 =  0.0;
sender_angular_vel[1]                 =  0.0;
sender_angular_vel[2]                 =  0.0;
```

```
    sender_angular_vel_PRESENT                  =    0;
    character_set                               = '\0';
    character_set_PRESENT                       =    0;
    markings[0]                                 = '\0';
    markings[1]                                 = '\0';
    markings[2]                                 = '\0';
    markings[3]                                 = '\0';
    markings[4]                                 = '\0';
    markings[5]                                 = '\0';
    markings[6]                                 = '\0';
    markings[7]                                 = '\0';
    markings[8]                                 = '\0';
    markings[9]                                 = '\0';
    markings[10]                                 = '\0';
    markings_PRESENT                            =    0;
    capabilities                                =    0;
    capabilities_PRESENT                        =    0;

}


entity_state_PDU::write_PDU()
{

// This is the hacked together code that allows the SDM/MP
// PDU Classes to use the DIS 2.0.3 Network Library.
//
// This is for testing purposes only and should be removed
// prior to using the OpenNPSNET network manager.

  EntityStatePDU epdu;

  // Set the present variables and set the bit mask.
  set_present();
  set_bit_field();

  //fill in the parameters of an entity state PDU, this assumes there
  //are no articulated parameters
  //ID and Type

  epdu.entity_state_header.protocol_version = protocol_version;
  epdu.entity_state_header.exercise_ident = exercise_ID;
  epdu.entity_state_header.type = PDU_Type;
  epdu.entity_state_header.padding_8 = protocol_family;
  epdu.entity_state_header.time_stamp = time_stamp;
  epdu.entity_state_header.length = length;
  epdu.entity_state_header.padding_16 = flag;  // stuff the flag here
                                               // so it fits!
  epdu.entity_id.address.site = sender_entityID[0];
  epdu.entity_id.address.host = sender_entityID[1];
  epdu.entity_id.entity = sender_entityID[2];
```

```
epdu.force_id = sender_force_ID;
epdu.num_articulat_params = num_articulation_params;

// Fill in the entity kind information
epdu.entity_type.entity_kind = entity_kind;
epdu.entity_type.domain = domain;
epdu.entity_type.country = country;
epdu.entity_type.category = category;
epdu.entity_type.subcategory = sub_category;
epdu.entity_type.specific = specific;
epdu.entity_type.extra = extra;

// Fill in the alternate entity kind information
if(alt_entity_kind_PRESENT == 1)
   epdu.alt_entity_type.entity_kind = alt_entity_kind;
if(alt_domain_PRESENT == 1)
epdu.alt_entity_type.domain = alt_domain;
if(alt_country_PRESENT == 1)
  epdu.alt_entity_type.country = alt_country;
if(alt_category_PRESENT == 1)
  epdu.alt_entity_type.category = alt_category;
if(alt_sub_category_PRESENT == 1)
  epdu.alt_entity_type.subcategory = alt_sub_category;
if(alt_specific_PRESENT == 1)
  epdu.alt_entity_type.specific = alt_specific;
if(alt_extra_PRESENT == 1)
  epdu.alt_entity_type.extra = alt_extra;

//How fast is it going
if(sender_linear_velocity_PRESENT == 1)
{
    epdu.entity_velocity.x = sender_linear_velocity[0];
    epdu.entity_velocity.y = sender_linear_velocity[1];
    epdu.entity_velocity.z = sender_linear_velocity[2];
}

//Where we are
epdu.entity_location.x = sender_location[0];
epdu.entity_location.y = sender_location[1];
epdu.entity_location.z = sender_location[2];

//How we are facing
epdu.entity_orientation.psi   = sender_orientation[0];
epdu.entity_orientation.theta = sender_orientation[1];
epdu.entity_orientation.phi   = sender_orientation[2];

//what we look like
if(sender_appearance_PRESENT == 1)
   epdu.entity_appearance = sender_appearance;
```

```
   //project our movement
   if(DeadReckAlgorithm_PRESENT == 1)
        epdu.dead_reckon_params.algorithm = DeadReckAlgorithm;

   //Send out dr accelerations and velocities
   if(sender_linear_accel_PRESENT == 1)
   {
      epdu.dead_reckon_params.linear_accel[0] = sender_linear_accel[0];
      epdu.dead_reckon_params.linear_accel[1] = sender_linear_accel[1];
      epdu.dead_reckon_params.linear_accel[2] = sender_linear_accel[2];
   }

   // We don't calculate these at the moment but for the future
   if(sender_angular_vel_PRESENT == 1)
   {
      epdu.dead_reckon_params.angular_velocity[0] =
sender_angular_vel[0];
      epdu.dead_reckon_params.angular_velocity[1] =
sender_angular_vel[1];
      epdu.dead_reckon_params.angular_velocity[2] =
sender_angular_vel[2];
   }

   if(character_set_PRESENT == 1)
      epdu.entity_marking.character_set = character_set;

   if(markings_PRESENT == 1)
     strncpy ((char *)epdu.entity_marking.markings,(char *)markings, 10);

   if(capabilities_PRESENT == 1)
      epdu.capabilities = capabilities;

   if ( net->write_pdu((char *)&epdu,EntityStatePDU_Type) == FALSE)
            printf("net_write() failed\n");

   cout << "Wrote Entity State PDU " << pdu_length()
        << " bytes long." << endl << endl;

   print_PDU();
   cout << endl << endl;
   return 0;
}

int entity_state_PDU::read_PDU()
{

// This is the hacked together code that allows the SDM/MP
// PDU Classes to use the DIS 2.0.3 Network Library.
//
// This is for testing purposes only and should be removed
// prior to using the OpenNPSNET network manager.
```

```
entity_state_PDU es_pdu;

char *pdu;
int swap_bufs;
EntityStatePDU *epdu;
PDUType type;
sender_info pdu_sender_info;

net->read_pdu(&pdu, &type, pdu_sender_info, swap_bufs);

// If there is a PDU process it based on the PDU type. We only care
// about ESPDUs. All others are ignored.
if ( pdu != NULL )
{
   switch(type)
   {
      case (EntityStatePDU_Type):
      epdu = (EntityStatePDU *)pdu;

      // Get the sender information
      //lookup_name_from_address(pdu_sender_info);

      // Stuff the ESPDU into an entity node if it is
      // from a platform or life form. Don't care
      // about munitions or other ESPDUs.
      if((epdu->entity_type.entity_kind == 1) ||
         (epdu->entity_type.entity_kind == 3))
      {
         // Create the stuff needed to read the bit mask.
         unsigned short mask = 1 << 15;
         unsigned short flag_check = 0;

         //fill in the parameters of an entity state PDU,
         //this assumes there
         //are no articulated parameters
         //ID and Type
         protocol_version = epdu->entity_state_header.
                            protocol_version;
         exercise_ID = epdu->entity_state_header.exercise_ident;
         PDU_Type = epdu->entity_state_header.type;
         protocol_family = epdu->entity_state_header.padding_8;
         time_stamp = epdu->entity_state_header.time_stamp;
         flag = epdu->entity_state_header.padding_16;
         flag_check = epdu->entity_state_header.padding_16;
         header_padding = 0;

         sender_entityID[0] = epdu->entity_id.address.site;
         sender_entityID[1] = epdu->entity_id.address.host;
         sender_entityID[2] = epdu->entity_id.entity;
```

```
sender_force_ID = epdu->force_id;
num_articulation_params = epdu->num_articulat_params;

// Fill in the entity kind information
entity_kind = epdu->entity_type.entity_kind;
domain = epdu->entity_type.domain;
country = epdu->entity_type.country;
category = epdu->entity_type.category;
sub_category = epdu->entity_type.subcategory;
specific = epdu->entity_type.specific;
extra = epdu->entity_type.extra;

// Fill in the alternate entity kind information

if(flag_check & mask ? 1 : 0)
{
   alt_entity_kind = epdu->alt_entity_type.entity_kind;
   alt_entity_kind_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
{
   alt_domain = epdu->alt_entity_type.domain;
   alt_domain_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
{
   alt_country = epdu->alt_entity_type.country;
   alt_country_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
{
   alt_category = epdu->alt_entity_type.category;
   alt_category_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
{
   alt_sub_category = epdu->alt_entity_type.subcategory;
   alt_sub_category_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
```

```
{
   alt_specific = epdu->alt_entity_type.specific;
   alt_specific_PRESENT = 1;
}
flag_check <<= 1;

if(flag_check & mask ? 1 : 0)
{
   alt_extra = epdu->alt_entity_type.extra;
   alt_extra_PRESENT = 1;
}
flag_check <<= 1;

//How fast is it going
if(flag_check & mask ? 1 : 0)
{
   sender_linear_velocity[0] = epdu->entity_velocity.x;
   sender_linear_velocity[1] = epdu->entity_velocity.y;
   sender_linear_velocity[2] = epdu->entity_velocity.z;
   sender_linear_velocity_PRESENT = 1;
}
flag_check <<= 1;

//Where we are
sender_location[0] = epdu->entity_location.x;
sender_location[1] = epdu->entity_location.y;
sender_location[2] = epdu->entity_location.z;

//How we are facing
sender_orientation[0] = epdu->entity_orientation.psi;
sender_orientation[1] = epdu->entity_orientation.theta;
sender_orientation[2] = epdu->entity_orientation.phi;

//what we look like
if(flag_check & mask ? 1 : 0)
{
   sender_appearance = epdu->entity_appearance;
   sender_appearance_PRESENT = 1;
}
flag_check <<= 1;

//project our movement
if(flag_check & mask ? 1 : 0)
{
   DeadReckAlgorithm = epdu->dead_reckon_params.algorithm;
   DeadReckAlgorithm_PRESENT = 1;
}
flag_check <<= 1;

//Send out dr accelerations and velocities
if(flag_check & mask ? 1 : 0)
```

```
            {
               sender_linear_accel[0] = epdu-
>dead_reckon_params.linear_accel[0];
               sender_linear_accel[1] = epdu-
>dead_reckon_params.linear_accel[1];
               sender_linear_accel[2] = epdu-
>dead_reckon_params.linear_accel[2];
               sender_linear_accel_PRESENT = 1;
            }
            flag_check <<= 1;

            // We don't calculate these at the moment but for the future
            if(flag_check & mask ? 1 : 0)
            {
               sender_angular_vel[0] = epdu->dead_reckon_params.
                                       angular_velocity[0];
               sender_angular_vel[1] = epdu->dead_reckon_params.
                                       angular_velocity[1];
               sender_angular_vel[2] = epdu->dead_reckon_params.
                                       angular_velocity[2];
               sender_angular_vel_PRESENT = 1;
            }
            flag_check <<= 1;

            if(flag_check & mask ? 1 : 0)
            {
               character_set = epdu->entity_marking.character_set;
               character_set_PRESENT = 1;
            }
            flag_check <<= 1;

            if(flag_check & mask ? 1 : 0)
            {
               strncpy ((char *)markings,
                        (char *)epdu->entity_marking.markings, 10);
               markings_PRESENT = 1;
            }
            flag_check <<= 1;

            if(flag_check & mask ? 1 : 0)
            {
               capabilities = epdu->capabilities;
               capabilities_PRESENT = 1;
            }
            flag_check <<= 1;

            length = pdu_length();

            // Print the PDU
            cout << "Entity State PDU of " << pdu_length()
                 << " bytes recieved." << endl << endl;
```

```
            print_PDU();
            cout << endl << endl;
         }

         break;

         default:
         break;
      }
   }
   return 0;
}

int entity_state_PDU::pdu_length()
{
   unsigned short length = 0;

   length += sizeof(protocol_version);
   length += sizeof(exercise_ID);
   length += sizeof(PDU_Type);
   length += sizeof(protocol_family);
   length += sizeof(time_stamp);
   length += sizeof(length);
   length += sizeof(header_padding);
   length += sizeof(sender_entityID[0]);
   length += sizeof(sender_entityID[1]);
   length += sizeof(sender_entityID[2]);
   length += sizeof(sender_force_ID);
   length += sizeof(num_articulation_params);
   length += sizeof(entity_kind);
   length += sizeof(domain);
   length += sizeof(country);
   length += sizeof(category);
   length += sizeof(sub_category);
   length += sizeof(specific);
   length += sizeof(extra);
   length += sizeof(sender_location[0]);
   length += sizeof(sender_location[1]);
   length += sizeof(sender_location[2]);
   length += sizeof(sender_orientation[0]);
   length += sizeof(sender_orientation[1]);
   length += sizeof(sender_orientation[2]);
   length += sizeof(flag);

   if(alt_entity_kind_PRESENT == 1)
   {
      length += sizeof(alt_entity_kind);
   }

   if(alt_domain_PRESENT == 1)
   {
```

```
    length += sizeof(alt_domain);
}

if(alt_country_PRESENT == 1)
{
    length += sizeof(alt_country);
}

if(alt_category_PRESENT == 1)
{
    length += sizeof(alt_category);
}

if(alt_sub_category_PRESENT == 1)
{
    length += sizeof(alt_sub_category);
}

if(alt_specific_PRESENT == 1)
{
    length += sizeof(alt_specific);
}

if(alt_extra_PRESENT == 1)
{
    length += sizeof(alt_extra);
}

if(sender_linear_velocity_PRESENT == 1)
{
    length += sizeof(sender_linear_velocity[0]);
    length += sizeof(sender_linear_velocity[1]);
    length += sizeof(sender_linear_velocity[2]);
}

if(sender_appearance_PRESENT == 1)
{
    length += sizeof(sender_appearance);
}

if(DeadReckAlgorithm_PRESENT == 1)
{
    length += sizeof(DeadReckAlgorithm);
}

if(OtherParams_PRESENT == 1)
{
    length += sizeof(OtherParams[0]);
    length += sizeof(OtherParams[1]);
    length += sizeof(OtherParams[2]);
    length += sizeof(OtherParams[3]);
```

```
        length += sizeof(OtherParams[4]);
        length += sizeof(OtherParams[5]);
        length += sizeof(OtherParams[6]);
        length += sizeof(OtherParams[7]);
        length += sizeof(OtherParams[8]);
        length += sizeof(OtherParams[9]);
        length += sizeof(OtherParams[10]);
        length += sizeof(OtherParams[11]);
        length += sizeof(OtherParams[12]);
        length += sizeof(OtherParams[13]);
        length += sizeof(OtherParams[14]);
    }

    if(sender_linear_accel_PRESENT == 1)
    {
        length += sizeof(sender_linear_accel[0]);
        length += sizeof(sender_linear_accel[1]);
        length += sizeof(sender_linear_accel[2]);
    }
    if(sender_angular_vel_PRESENT == 1)
    {
        length += sizeof(sender_angular_vel[0]);
        length += sizeof(sender_angular_vel[1]);
        length += sizeof(sender_angular_vel[2]);
    }
    if(character_set_PRESENT == 1)
    {
        length += sizeof(character_set);
    }

    if(markings_PRESENT == 1)
    {
        length += sizeof(markings[0]);
        length += sizeof(markings[1]);
        length += sizeof(markings[2]);
        length += sizeof(markings[3]);
        length += sizeof(markings[4]);
        length += sizeof(markings[5]);
        length += sizeof(markings[6]);
        length += sizeof(markings[7]);
        length += sizeof(markings[8]);
        length += sizeof(markings[9]);
        length += sizeof(markings[10]);
    }
    if(capabilities_PRESENT == 1)
    {
        length += sizeof(capabilities);
    }
    return length;
}
```

```
void entity_state_PDU::set_bit_field()
{
   unsigned short mask = 0;

   if(alt_entity_kind_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 15;
      flag |= mask;
   }

   if(alt_domain_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 14;
      flag |= mask;
   }

   if(alt_country_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 13;
      flag |= mask;
   }

   if(alt_category_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 12;
      flag |= mask;
   }

   if(alt_sub_category_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 11;
      flag |= mask;
   }

   if(alt_specific_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 10;
      flag |= mask;
   }

   if(alt_extra_PRESENT == 1)
   {
      mask = 0;
      mask = 1 << 9;
      flag |= mask;
```

```
}

if(sender_linear_velocity_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 8;
   flag |= mask;
}

if(sender_appearance_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 7;
   flag |= mask;
}

if(DeadReckAlgorithm_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 6;
   flag |= mask;
}

if(OtherParams_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 5;
   flag |= mask;
}

if(sender_linear_accel_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 4;
   flag |= mask;
}

if(sender_angular_vel_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 3;
   flag |= mask;
}

if(character_set_PRESENT == 1)
{
   mask = 0;
   mask = 1 << 2;
   flag |= mask;
}
```

```
    if(markings_PRESENT == 1)
    {
       mask = 0;
       mask = 1 << 1;
       flag |= mask;
    }

    if(capabilities_PRESENT == 1)
    {
       mask = 0;
       mask = 1 << 0;
       flag |= mask;
    }


}

void entity_state_PDU::print_PDU()
{
    set_bit_field();

    cout << "protocol_version = " << (int)protocol_version << endl;
    cout << "exercise_ID = " << (int)exercise_ID << endl;
    cout << "PDU_Type = " << (int)PDU_Type << endl;
    cout << "protocol_family = " << (int)protocol_family << endl;
    cout << "time_stamp = " << time_stamp << endl;
    cout << "length = " << length << endl;
    cout << "header_padding = " << header_padding << endl;
    cout << "sender_entityID[0] = " << sender_entityID[0] << endl;
    cout << "sender_entityID[1] = " << sender_entityID[1] << endl;
    cout << "sender_entityID[2] = " << sender_entityID[2] << endl;
    cout << "sender_force_ID = " << (int)sender_force_ID << endl;
    cout << "num_articulation_params = "
         << (int)num_articulation_params << endl;
    cout << "entity_kind = " << (int)entity_kind << endl;
    cout << "domain = " << (int)domain << endl;
    cout << "country = " << country << endl;
    cout << "category = " << (int)(int)category << endl;
    cout << "sub_category = " << (int)sub_category << endl;
    cout << "specific = " << (int)specific << endl;
    cout << "extra = " << (int)extra << endl;
    cout << "sender_location[0] = " << sender_location[0] << endl;
    cout << "sender_location[1] = " << sender_location[1] << endl;
    cout << "sender_location[2] = " << sender_location[2] << endl;
    cout << "sender_orientation[0] = " << sender_orientation[0] << endl;
    cout << "sender_orientation[1] = " << sender_orientation[1] << endl;
    cout << "sender_orientation[2] = " << sender_orientation[2] << endl;

    display_bits(flag);
```

```
if(alt_entity_kind_PRESENT == 1)
{
   cout << "alt_entity_kind = " << (int)alt_entity_kind << endl;
}
else
{
   cout << "alt_entity_kind is not Present!" << endl;
}
if(alt_domain_PRESENT == 1)
{
   cout << "alt_domain = " << (int)alt_domain << endl;
}
else
{
   cout << "alt_domain is not Present!" << endl;
}
if(alt_country_PRESENT == 1)
{
   cout << "alt_country = " << alt_country << endl;
}
else
{
   cout << "alt_country is not Present!" << endl;
}
if(alt_category_PRESENT == 1)
{
   cout << "alt_category = " << (int)alt_category << endl;
}
else
{
   cout << "alt_category is not Present!" << endl;
}
if(alt_sub_category_PRESENT == 1)
{
   cout << "alt_sub_category = " << (int)alt_sub_category << endl;
}
else
{
   cout << "alt_sub_category is not Present!" << endl;
}
if(alt_specific_PRESENT == 1)
{
   cout << "alt_specific = " << (int)alt_specific << endl;
}
else
{
   cout << "alt_specific is not Present!" << endl;
}
if(alt_extra_PRESENT == 1)
{
   cout << "alt_extra = " << (int)alt_extra << endl;
```

```
}
else
{
   cout << "alt_extra is not Present!" << endl;
}
if(sender_linear_velocity_PRESENT == 1)
{
   cout << "sender_linear_velocity[0] = "
        << sender_linear_velocity[0] << endl;
   cout << "sender_linear_velocity[1] = "
        << sender_linear_velocity[1] << endl;
   cout << "sender_linear_velocity[2] = "
        << sender_linear_velocity[2] << endl;
}
else
{
   cout << "sender_linear_velocity is not Present!" << endl;
}
if(sender_appearance_PRESENT == 1)
{
   cout << "sender_appearance = " << sender_appearance << endl;
}
else
{
   cout << "sender_appearance is not Present!" << endl;
}
if(DeadReckAlgorithm_PRESENT == 1)
{
   cout << "DeadReckAlgorithm = " << (int)DeadReckAlgorithm << endl;
}
else
{
   cout << "DeadReckAlgorithm is not Present!" << endl;
}
if(OtherParams_PRESENT == 1)
{
   cout << "OtherParams[0] = " << (int)OtherParams[0] << endl;
   cout << "OtherParams[1] = " << (int)OtherParams[1] << endl;
   cout << "OtherParams[2] = " << (int)OtherParams[2] << endl;
   cout << "OtherParams[3] = " << (int)OtherParams[3] << endl;
   cout << "OtherParams[4] = " << (int)OtherParams[4] << endl;
   cout << "OtherParams[5] = " << (int)OtherParams[5] << endl;
   cout << "OtherParams[6] = " << (int)OtherParams[6] << endl;
   cout << "OtherParams[7] = " << (int)OtherParams[7] << endl;
   cout << "OtherParams[8] = " << (int)OtherParams[8] << endl;
   cout << "OtherParams[9] = " << (int)OtherParams[9] << endl;
   cout << "OtherParams[10] = " << (int)OtherParams[10] << endl;
   cout << "OtherParams[11] = " << (int)OtherParams[11] << endl;
   cout << "OtherParams[12] = " << (int)OtherParams[12] << endl;
   cout << "OtherParams[13] = " << (int)OtherParams[13] << endl;
   cout << "OtherParams[14] = " << (int)OtherParams[14] << endl;
```

```
}
else
{
   cout << "OtherParams is not Present!" << endl;
}
if(sender_linear_accel_PRESENT == 1)
{
   cout << "sender_linear_accel[0] = " << sender_linear_accel[0]
        << endl;
   cout << "sender_linear_accel[1] = " << sender_linear_accel[1]
        << endl;
   cout << "sender_linear_accel[2] = " << sender_linear_accel[2]
        << endl;
}
else
{
   cout << "sender_linear_accel is not Present!" << endl;
}
if(sender_angular_vel_PRESENT == 1)
{
   cout << "sender_angular_vel[0] = " << sender_angular_vel[0]
        << endl;
   cout << "sender_angular_vel[1] = " << sender_angular_vel[1]
        << endl;
   cout << "sender_angular_vel[2] = " << sender_angular_vel[2]
        << endl;
}
else
{
   cout << "sender_angular_vel is not Present!" << endl;
}
if(character_set_PRESENT == 1)
{
   cout << "character_set = " << (int)character_set << endl;
}
else
{
   cout << "character_set is not Present!" << endl;
}
if(markings_PRESENT == 1)
{
   cout << "markings[0] = " << markings[0] << endl;
   cout << "markings[1] = " << markings[1] << endl;
   cout << "markings[2] = " << markings[2] << endl;
   cout << "markings[3] = " << markings[3] << endl;
   cout << "markings[4] = " << markings[4] << endl;
   cout << "markings[5] = " << markings[5] << endl;
   cout << "markings[6] = " << markings[6] << endl;
   cout << "markings[7] = " << markings[7] << endl;
   cout << "markings[8] = " << markings[8] << endl;
   cout << "markings[9] = " << markings[9] << endl;
```

```
        cout << "markings[10] = " << markings[10] << endl;
    }
    else
    {
        cout << "markings is not Present!" << endl;
    }
    if(capabilities_PRESENT == 1)
    {
        cout << "capabilities = " << capabilities << endl;
    }
    else
    {
        cout << "capabilities is not Present!" << endl;
    }

}

void entity_state_PDU::display_bits(unsigned short value)
{
    int length = (sizeof(unsigned short) * 8);
    unsigned short cx, displayMask = 1 << 15;

    for(cx = 1; cx <= length; cx++)
    {
        cout << (value & displayMask ? '1' : '0');
        value <<= 1;

    }

    cout << endl;
}

void entity_state_PDU::set_present()
{
    if(alt_entity_kind != '\0')
    {
        alt_entity_kind_PRESENT = 1;
    }
    if(alt_domain != '\0')
    {
        alt_domain_PRESENT = 1;
    }
    if(alt_country != 0)
    {
        alt_country_PRESENT = 1;
    }
    if(alt_category != '\0')
    {
        alt_category_PRESENT = 1;
    }
    if(alt_sub_category != '\0')
```

```
{
   alt_sub_category_PRESENT = 1;
}
if(alt_specific != '\0')
{
   alt_specific_PRESENT = 1;
}
if(alt_extra != '\0')
{
   alt_extra_PRESENT = 1;
}
if(sender_linear_velocity[0] < 0.0 ||
   sender_linear_velocity[0] > 0.0 )
{
   sender_linear_velocity_PRESENT = 1;
}
if(sender_linear_velocity[1] < 0.0 ||
   sender_linear_velocity[1] > 0.0 )
{
   sender_linear_velocity_PRESENT = 1;
}
if(sender_linear_velocity[2] < 0.0 ||
   sender_linear_velocity[2] > 0.0 )
{
   sender_linear_velocity_PRESENT = 1;
}
if(sender_appearance != 0)
{
   sender_appearance_PRESENT = 1;
}
if(DeadReckAlgorithm != '\0')
{
   DeadReckAlgorithm_PRESENT = 1;
}
if(OtherParams[0] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[1] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[2] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[3] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[4] != '\0')
```

```
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[5] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[6] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[7] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[8] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[9] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[10] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[11] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[12] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[13] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(OtherParams[14] != '\0')
{
   OtherParams_PRESENT = 1;
}
if(sender_linear_accel[0] < 0.0 || sender_linear_accel[0] > 0.0 )
{
   sender_linear_accel_PRESENT = 1;
}
if(sender_linear_accel[1] < 0.0 || sender_linear_accel[1] > 0.0 )
{
   sender_linear_accel_PRESENT = 1;
}
```

```
if(sender_linear_accel[2] < 0.0 || sender_linear_accel[2] > 0.0 )
{
   sender_linear_accel_PRESENT = 1;
}
if(sender_angular_vel[0] < 0.0 || sender_angular_vel[0] > 0.0 )
{
   sender_angular_vel_PRESENT = 1;
}
if(sender_angular_vel[1] < 0.0 || sender_angular_vel[1] > 0.0 )
{
   sender_angular_vel_PRESENT = 1;
}
if(sender_angular_vel[2] < 0.0 || sender_angular_vel[2] > 0.0 )
{
   sender_angular_vel_PRESENT = 1;
}
if(character_set != '\0')
{
   character_set_PRESENT = 1;
}
if(markings[0] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[1] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[2] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[3] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[4] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[5] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[6] != '\0')
{
   markings_PRESENT = 1;
}
if(markings[7] != '\0')
{
   markings_PRESENT = 1;
```

```
    }
    if(markings[8] != '\0')
    {
       markings_PRESENT = 1;
    }
    if(markings[9] != '\0')
    {
       markings_PRESENT = 1;
    }
    if(markings[10] != '\0')
    {
       markings_PRESENT = 1;
    }
    if(capabilities != 0)
    {
       capabilities_PRESENT = 1;
    }

}


// Function: ==
// Purpose:  Overloaded equality comparison operator

int entity_state_PDU::operator==(const entity_state_PDU & e)
{
            if ( (sender_entityID[0] == 0) ||
                        (sender_entityID[0] == e.sender_entityID[0]) )
                    {
                    if ( (sender_entityID[1] == 0) ||
                                (sender_entityID[1] ==
e.sender_entityID[1]) )
                                {
                                if ( (sender_entityID[2] == 0) ||
                                            (sender_entityID[2] ==
e.sender_entityID[2]) )
                                        return 1;
                                }
                    }
            return 0;
}


// Function: !=
// Purpose:  Overloaded inequality comparison operator

int entity_state_PDU::operator!=(const entity_state_PDU & e)
{
            if ( (sender_entityID[0] == 0) ||
                        (sender_entityID[0] == e.sender_entityID[0]) )
                    {
```

```
                        if ( (sender_entityID[1] == 0) ||
                                    (sender_entityID[1] ==
e.sender_entityID[1]) )
                                    {
                                    if ( (sender_entityID[2] == 0) ||
                                                (sender_entityID[2] ==
e.sender_entityID[2]) )
                                                return 0;
                                    }
                        }
            return 1;
}
```

LIST OF REFERENCES

BRU95 Brutzman, Donald, Macedonia, Michael and Zyda, Michael, *Internetwork Infrastructure Requirements for virtual Environments*, submitted to the First VRML Symposium, 1995.

CANTER95 Canterbury, Michael, *An Automated Approach to Distributed Interactive Simulation (DIS) Protocol Entity Development*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.

COHEN94a Cohen, Danny, *NG-DIS-PDU: The Next Generation of DIS-PDU,* Proceedings of the 10th DIS Conference, Orlando, Florida, March 1994.

COHEN94b Cohen, Danny, *DIS - Back to Basics,* Proceedings of the 11th DIS Conference, Orlando, Florida, September 1994.

COHEN95 Cohen, Danny and Kirsh, Moshe, *SDM/MP: Self Defined Messages with Multiple Presentations,* Proceedings of the 13th DIS Conference, Orlando, Florida, September 1995.

COHEN96a Cohen, Danny and Kirsh, Moshe, *A SDM/MP Approach to DIS-3,* Proceedings of the 14th DIS Conference, Orlando, Florida, March 1996.

COHEN96b Cohen, Danny and Kirsh, Moshe, *A Proposal for DIS++ (Based on SDM/MP),* Perceptronics, Woodland Hills, California, March 1996.

DIS94 DIS Steering Committee, *The DIS Vision - A Map to the Future of Distributed Simulation*, Institute for Simulation and Training, Orlando, Florida, May 1994.

DURLACH95 Durlach, Nathaniel I. and Mavor, Anne S., *Virtual Reality: Scientific and Technological Challenges,* National Academy Press, Washington, D.C. 1995.

IEEE1278 Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 1278-1993, *Standard for Information Technology, Protocols for Distributed Interactive Simulation,* March 1993.

LEVI92 Levine, John R., Mason, Tony, Brown, Doug, *LEX & YACC*, O'Reilly & Associates, Sebastopol, California, 1992.

MAC94 Macedonia, Michael R., Zyda, Michael J., Pratt, David R., Barham, Paul T., Zeswitz, Steven, *NPSNET: A Network Software Architecture for Large Scale Virtual Environments*, Presence, 3, 4, Winter 1994.

MAC95a Macedonia, Michael R., *A Network Architecture for Large Scale Virtual Environments,* Dissertation, Naval Postgraduate School, Monterey, California, June 1995.

MAC95b Macedonia, Michael R., Zyda, Michael J., Pratt, David R., Barham, Paul T., *Exploiting Reality with Multicast Groups,* IEEE Computer Graphics and Applications, September 1995.

MOR95 Morrison, John, *The VR-Link, Networked Virtual Environment Software Infrastructure*, Presence, 4,2, Summer 95.

SCHUG95 Schug, Klaus H., *DIS NG - A Flexible Protocol for all Simulation Applications*, Proceedings of the 13th DIS Conference, Orlando, 1995.

WHITE95 White, James E., *Telescript Technology: Mobile Agents*, General Magic White Paper, Mountain View California, 1995.

ZES93 Zeswitz, Steven R., *NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Ted Lewis, Code CS/Lt Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
4.	Dr. Michael Zyda, Code CS/Zk Computer Science Department Naval Postgraduate School Monterey, CA 93943	6
5.	Dr. Don Brutzman, Code UW/Br Interdisciplinary Academic Group Naval Postgraduate School Monterey, CA 93943	1
6.	Prof John Falby, Code CS/Fa Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
7.	CPT Steven W. Stone 9810 Christina Drive Riverview, FL 33569	1