STARS

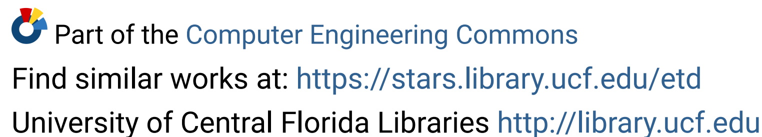
University of Central Florida
STARS

Electronic Theses and Dissertations, 2004-2019

2004

Data Transmission Scheduling For Distributed Simulation Using Packet A

Juan Vargas-Morales
University of Central Florida



Part of the Computer Engineering Commons
Find similar works at: https://stars.library.ucf.edu/etd
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Vargas-Morales, Juan, "Data Transmission Scheduling For Distributed Simulation Using Packet A" (2004). *Electronic Theses and Dissertations, 2004-2019*. 255.
https://stars.library.ucf.edu/etd/255

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

Data Transmission Scheduling for Distributed Simulation Using Packet Alloying

by

Juan José Vargas-Morales
B.S. Universidad de Costa Rica, 1977
M.S. University of Delaware, 1991

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2004

Major Professor:
Ronald F. DeMara

Abstract

Communication bandwidth and latency reduction techniques are developed for Distributed Interactive Simulation (DIS) protocols. Using logs from vignettes simulated by the OneSAF Testbed Baseline (OTB), a discrete event simulator based on the OMNeT++ modeling environment is developed to analyze the Protocol Data Unit (PDU) traffic over a wireless flying Local Area Network (LAN). Alternative PDU bundling and compression techniques are studied under various metrics including *slack time*, *travel time*, *queue length*, and *collision rate*. Based on these results, *Packet Alloying*, a technique for the optimized bundling of packets, is proposed and evaluated.

Packet Alloying becomes more active when it is needed most: during negative spikes of transmission slack time. It produces aggregations that preserve the internal PDU format, allowing the resulting packets to be subjectable to further bundling and/or compression by conventional techniques. To optimize the selection of bundle delimitation, three online predictive strategies were developed: *Neural-Network* based, *Always-Wait*, and *Always-Send*. These were compared with three offline strategies defined as *Type*, *Type-Length* and *Type-Length-Size*. Applying *Always-Wait* to the studied vignette using the wireless links set to 64 Kbps, a reduction in the magnitude of negative slack time from -75 to -9 seconds for the worst spike was achieved, which represents a reduction of 88 %. Similarly, at 64 Kbps, *Always-Wait* reduced the average satellite queue length from 2,963 to 327 messages for a 89 % reduction. From the analysis of negative slack-time spikes it was determined which PDU types are of highest priority. The router and satellite queues in the

case study were modified accordingly using a priority-based transmission scheduler. The analysis of total travel times based of PDU types numerically shows the benefit obtained.

The contributions of this dissertation include the formalization of a selective PDU bundling scheme, the proposal and study of different predictive algorithms for the next PDU, and priority-based optimization using Head-of-Line (HoL) service. These results demonstrate the validity of packet optimizations for distributed simulation environments and other possible applications such as TCP/IP transmissions.

To my wife Vilma, my son Daniel, and my mother

Acknowledgments

In the first place, I want to give thanks to God because He supported me during these four years at UCF, and protected me when the hurricanes came over Florida and over my home.

I would like to thank my advisor, Dr. Ronald DeMara, for his many suggestions and constant support during this research. His detailed revision of the manuscript gave this document the quality required for a doctoral dissertation.

I am also thankful to Dr. Avelino González for his friendship and guidance through the early years at UCF. He was the contact that made possible this adventure.

Of course, I am grateful to my wife Vilma for her patience and love during these four years. She supported me on uncountable occasions and encouraged me to finish this endeavor. Thanks also to my son Daniel that did his part making my life a lot more enjoyable, relieving me from the hard work from time to time.

Thanks to PEO STRI because this work was supported in part by the U.S. Army Research Development and Engineering Command (RDE-COMM) as part of the Embedded Combined Arms Team Trainer and Mission Rehearsal (ECATT-MR) Science and Technology Objective (STO) contract N61339-02-C-0097.

I should also thank the University of Costa Rica that allowed me to temporarily leave my work there while pursuing studies in Florida.

Without all this help, this effort would never have come to fruition.

Table of Contents

List of Tables . . . xii

List of Figures . . . xiii

List of Acronyms . . . xvii

CHAPTER 1: INTRODUCTION . . . 1

1.1 Overview . . . 1

1.2 Distributed Simulation Environments . . . 4

1.3 Need for Communication Optimizations . . . 6

1.4 Outline of Dissertation . . . 8

1.5 Contributions of Dissertation . . . 9

CHAPTER 2: PREVIOUS WORK . . . 12

2.1 Bundling and Aggregation of Network Packets . . . 12

2.2 Data Compression Techniques . . . 16

2.3 Data Transmission Optimizations . . . 20

2.4 Comparison of Bundling Strategies . . . 21

CHAPTER 3: COMMUNICATION RESOURCES AND ARCHITECTURE . . . 24

3.1 EMPR Simulation Vignette . . . 25

3.2 ES Communication Architecture . . . 28

3.3 Transmitting and Receiving Devices 31
3.3.1 Primitive Modules in OMNeT 33
3.3.2 Compound Module Definition 39
3.3.3 Flying LAN Computer Nodes 40
3.3.4 Instantiation of the Network 44
3.4 Messages in the Simulator 45

CHAPTER 4: PACKET ALLOYING BUNDLING TECHNIQUE 48

4.1 Characteristics of Embedded Simulation Traffic Impacting Bundling . 48
4.1.1 Real-Time Dynamics 49
4.1.2 High Ratio of ESPDU Traffic 49
4.1.3 Low Volume of High Priority PDUs 50
4.1.4 Transmission Redundancy 50
4.1.5 Regular Packet Structure 51
4.1.6 Broadcast Transmission 51
4.1.7 Low Bandwidth 52
4.1.8 Simultaneous Scheduling of PDU Bursts 52
4.2 Offline and Online Algorithms 52
4.3 Packet Alloying Modes 56
4.3.1 Mathematical Description of Alloying 59
4.3.2 Online Bundling Strategies 61
4.3.3 Offline Bundling Strategy 62
4.4 Development of Alloying in Simulation Model 65

CHAPTER 5: EMBEDDED SIMULATION TRAFFIC ANALYSIS 73

5.1 Processing Flow and Sequencing 73

5.2 Input Data and AWK preprocessing 74

5.2.1 PDU Example 1 77

5.2.2 PDU Example 2 78

5.3 Transmission Parameters Analyzed 80

5.3.1 Architecture-Independent Bandwidth Analysis 81

5.3.2 Analysis of Simulation Results 86

5.4 Simulation ST: Vignette With Single Transmitter 90

5.4.1 Independent Analysis of Logged PDUs 90

5.4.2 Slack Time 93

5.4.3 Travel Time 94

5.4.4 Queue Length 95

5.4.5 Collisions 96

5.4.6 Conclusions of Simulation ST 96

5.5 Simulation DT: Vignette with Dual Transmitters 97

5.5.1 Independent Analysis of Logged PDUs 98

5.5.2 Slack Time 101

5.5.3 Travel Time 102

5.5.4 Queue Length 105

5.5.5 Collisions 107

5.5.6 Conclusions of Simulation DT 109

5.6 Simulation MR1T6: Vignette MR1 with Six Transmitters 109

5.6.1 Independent Analysis of Logged PDUs . . . 110
5.6.2 Slack Time . . . 113
5.6.3 Travel Time . . . 117
5.6.4 Queue Length . . . 119
5.6.5 Collisions . . . 123
5.6.6 Spike Analysis of Slack Time . . . 125
5.6.7 Conclusions of Simulation MR1T6 . . . 129
5.7 Simulation MR1GS: Vignette MR1 Revisited . . . 131
5.7.1 Independent Analysis and PDU Assignment . . . 132
5.7.2 Slack Time . . . 132
5.7.3 Travel Time . . . 135
5.7.4 Queue Length . . . 136
5.7.5 Collisions . . . 139
5.7.6 Conclusions of Simulation MR1GS . . . 140
5.8 Simulation using Head-of-Line Strategy . . . 141

CHAPTER 6: TRAFFIC OPTIMIZATION USING PACKET ALLOYING . . . 147

6.1 Input Data Logs in Simulation MR1PA . . . 148
6.2 Slack Time Analysis . . . 148
6.3 Travel Time Analysis . . . 152
6.4 Queue Length Analysis . . . 155
6.5 Collision Accumulation . . . 158
6.6 Conclusions of Packet Alloying Simulation . . . 159

CHAPTER 7: CONCLUSION **161**

7.1 Summary of Accomplishments 162

7.2 Simulations Performed 163

7.3 Architecture-Independent Analysis 165

7.4 Conclusions about OTB Traffic 166

7.5 Packet Alloying 168

7.6 HoL Priority Service 171

7.7 Future Work 172

APPENDIX A: MR1 VIGNETTE **175**

A.1 Background 176

A.2 General Vignette Description 178

A.2.1 Situation and Mission Prior to Start of Vignette 179

A.2.2 The 1st UA Prepares for Entry Operations 180

A.3 Specific Vignette for Project 181

APPENDIX B: NED SOURCE CODE **186**

B.1 File Generator.ned 187

B.2 File Router.ned 187

B.3 File Satellite.ned 188

B.4 File Simplebus.ned 188

B.5 File Sink.ned 190

B.6 File TheNet.ned 190

B.7 File Omnetpp.ini 196

APPENDIX C: AWK SOURCE CODE **198**

C.1 AWK Script for PDU Parsing 199

C.2 AWK Script for Independent Analysis 202

C.3 AWK Scripts for Neural Network Processing 203

APPENDIX D: SIMULATOR SOURCE CODE **206**

LIST OF REFERENCES **280**

List of Tables

1 Comparison of Bundling Techniques 21

2 Routing Table in Broadcast Mode 37

3 Comparison of Two Consecutive `Po_fire_parameters` PDUs 58

4 Types of PDUs and Volume of Bytes Transmitted for Each Type . . . 110

5 Packets With Positive Slack at Sending Sites in Simulation MR1T6 . 116

6 Collisions per Second in Simulation MR1T6 131

7 PDU Priorities for HoL Service 142

8 Travel Time by Priority at Plane 7 145

9 Travel Time for CONUS PDUs by Priority at Plane 7 145

10 Slack Time for All of the Algorithms at Ground Station 151

11 Average and Standard Deviation of Travel Time Measured at Sink 0 . 154

12 Total Travel time at sink 21 (64 Kbps, 256 Kbps) 155

13 Satellite Queue Length for Various Algorithms and Bandwidths . . . 157

List of Figures

1 The Flying Network 28

2 Communication Architecture Model 29

3 OMNeT Screenshot of the Entire Network 32

4 Source File Generator.ned 34

5 File Simplebus.ned 35

6 File Sink.ned 36

7 Router Onboard a Plane and its Connections 37

8 File Router.ned 38

9 File Satellite.ned 39

10 OMNeT Representation of a Computer Node and its Components 40

11 Ned Code of a Computer Node 41

12 Airplane View Showing 3 Computer Nodes, a Bus and a Router 42

13 OMNeT View of the Ground Station and its Components 42

14 Ned Code of Ground Station 43

15 Instantiation of the Network `TheNet` 44

16 Initialization File Omnetpp.ini 46

17 Pseudo-Algorithm of PDU Bundling 67

18 Decision Tree of the Algorithm Used by Generators 70

19 Overview of the Simulation Process 75
20 Sample of the Contents of Summary File `Datan.txt` 76
21 Complete PDU of Type `Po_variable` 79
22 Short PDU of Type *Acknowledge* 80
23 PDU Type Distribution Generated in Simulation ST 91
24 Minimum Bandwidth Requirements 92
25 Slack Time to Send Next Message at Generator 0 (64 Kbps) 93
26 Travel Time as Sensed by Ground Station (64 Kbps) 94
27 Messages in Router 0 in Plane 0 (64 Kbps) 95
28 Messages in the Satellite (64 Kbps) 96
29 PDU Type Distribution Generated in Simulation DT 99
30 Minimum Bandwidth Requirements 100
31 Slack Time at Plane 0 and Ground Station (64 Kbps) 101
32 Slack Time to Send Next Message at Plane 0 (64 Kbps) 102
33 Slack Time at Plane 0 and Ground Station (400 Kbps) 103
34 Travel Time at Plane 0 and Ground Station (64 Kbps) 103
35 Travel Time at Plane 7 (64 Kbps) 104
36 Travel Times at Plane 7 Zoomed In (400 Kbps) 105
37 Comparison of Queue Lengths of Plane 0 and Satellite (64 Kbps) . . 106
38 Comparison of Queue Lengths of Plane 0 and satellite (400 Kbps) . . 106
39 Collisions per Second Detected at Plane 1 (64 Kbps) 107
40 Collision Accumulation Over Time at Plane 7 (64 Kbps) 108
41 Distribution of PDUs in the Simulation of MR1 Vignette 111

42 Minimum Bandwidth Requirements in Simulation MR1T6 113
43 Slack Time of Generators (64 Kbps) 114
44 Zoom in of Slack Time at Ground Station (64 Kbps) 115
45 Zoom In of Slack Time at Ground Station (1,024 Kbps) 116
46 Travel Time at Node 2 in Plane 0 (64 Kbps) 117
47 Travel Time at Ground Station (64 Kbps) 118
48 Zoom In of Travel Times at Ground Station (64, 256 Kps) 120
49 Messages in System at Plane 0 (64 Kbps) 121
50 Messages in System at Plane 3 (64 Kbps) 121
51 Messages in System at Satellite (64 Kbps) 122
52 Zoom In of Messages in System at Plane 0 (64 Kbps and 256 Kbps) . 122
53 Zoom in of Messages in System at Satellite (64 Kbps and 256 Kbps) . 123
54 Collision Accumulation at Planes 1, 2 and 7 (64 Kbps) 124
55 Collisions in WSP Channel at Plane 7 (64 Kbps) 125
56 Collision Accumulation in Plane 7, Simulation MR1T6 126
57 Slack Time at Ground Station Showing Negative Spikes (64 Kbps) . . 127
58 Negative Spike at Second 1420 Showing Participating PDUs 128
59 Negative Spike at Second 1454 Showing Participating PDUs 130
60 Slack Time at Planes 0, 1, 2, 3, 4 and Ground Station (64 Kbps) . . . 133
61 Slack Time to Send Next Message at Ground Station (128 Kbps) . . 134
62 Zoom In of Slack Time at Ground Station, 256 Kbps 134
63 Travel Time at Plane 7 (64 Kbps) 135
64 Travel Time at Plane 7 (256 Kbps) 136

65 Messages in System at Plane 0 (64 Kbps) 137

66 Messages in System at the Satellite (64 Kbps and 256 Kbps) 138

67 Messages in System at the Satellite (1,024 Kbps) 139

68 Collision Accumulation at Plane 7 (64, 256, 512, 1,024 Kbps) 140

69 Effect of HoL on the Travel Time at Plane 7 (64 Kbps) 144

70 Slack Time at Ground Station for 6 Strategies (64 Kbps) 149

71 Comparison of Negative Slack for the Four Best Algorithms (64 Kbps) 150

72 Travel Time for the *Always-Wait* Strategy (64 Kbps and 128 Kbps) . 152

73 Close-up of Travel Time at Sink 0 in Plane 0 (128 Kbps) 153

74 Messages in Satellite (64 Kbps and 128 Kbps) 156

75 Preferred PDU Bundling Strategy 157

76 Collision Accumulation at Plane 7 (64, 256, 512, 1,024 Kbps) 159

77 Overall View of Theater of Operations 182

78 Details of Attack on Defensive Positions of ANFRA 183

79 Details of Advances on the Defensive Positions After Mortal Fire . . . 185

List of Acronyms

ACK	ACKnowledge
API	Application Program Interface
ARPA	Advanced Research Projects Agency
BBS	Battalion Battle Simulation
BLB	Battalion Level Behavior
bps	bits per second
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance
CCTT-SAF	Close Combat Tactical Trainer Semi-Automated Forces
CGF	Computer Generated Forces
CONUS	Continental United States
CPU	Central Processing Unit
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
DARPA	Defense Advanced Research Projects Agency
DIS	Distributed Interactive Simulation
DIS	Distributed Interactive Simulation
DDU	Differential Data Unit
DKDU	Differential Key Data Unit
DoD	Department of Defense
DPCM	Differential Pulse-Code Modulation
EMPR	En-route Mission Planning and Rehearsal
EMPRS	Enroute Mission Planning and Rehearsal Systems

EO	Embedded Operations
ES	Embedded Simulation
ET	Embedded Training
ESPDU	Entity State Protocol Data Unit
FCS	Future Combat System
FIFO	First In First Out
INVEST	Inter-Vehicle Embedded Simulation Technology
HITL	Human-In-The-Loop
HLA	High Level Architecture
HoL	Head-of-Line
JEF	Joint Experimental Federation
JEMPRS	Joint Enroute Mission Planning and Rehearsal System
JEMPRS-NT	Joint Enroute Mission Planning and Rehearsal System Near-Term
Kbps	Kilo bits per second
LAN	Local Area Network
LBRM	Log-Based Receiver-reliable Multicast
LRU	Least Recently Used
LZ	Lempel-Ziv algorithm
MAC	Medium Access Control
Mbps	Mega bits per second
ModSAF	Modular Semi-Automated Forces
MPQ	Multilevel Priority Queues
NED	Network topology Description language
NN	Neural Network
OFET	Objective Force Embedded Training
OMNeT++	Objective Modular Network Testbed in C++
OneSAF	One Semi-Automated Forces

OPFOR	Opposing Forces
OTB	OneSAF Testbed Baseline
O&O	Operations and Organizations
PDU	Protocol Data Unit
PEO STRI	(U.S. Army) Program Executive Office for Simulation, Training, & Instrumentation
PICA	Protocol Independent Compression Algorithm
QES	Quiescent Entity Service
QoS	Quality of Service
RPC	Remote Procedure Call
SAF	Semi-Automated Forces
SIMNET	SIMulation NETwork
SMS	Switch-Memory-Switch
STOW-E AG	Synthetic Theater of War–Europe Application Gateway
SPQ	Single Priority Queues
STRICOM	U. S. Army Simulation, Training and Instrumentation Command
STOW-E AG	Synthetic Theater of War-Europe Application Gateway
TCP/IP	Transmission Control Protocol / Internet Protocol
TDM	Time Division Multiplexing
UA	Unit of Action
UDP	User Datagram Protocol
UTC	Universal Coordinated Time
WAN	Wide Area Network
WGS	Wireless Ground station to Satellite link
WPP	Wireless Plane to Plane link
WSP	Wireless Satellite to Plane link

CHAPTER 1

INTRODUCTION

Computer modeling and simulation are commonly used in areas such as analysis and prediction of behavior of complex systems, training, education, computer games, etc., and have been applied to systems in many scientific disciplines such as Physics, Chemistry, Engineering, Psychology, Sociology, Meteorology, and others. When simulations are distributed over multiple processing nodes, important tradeoffs between communication and computation result. Factors impacting these tradeoffs are identified and defined below.

1.1 Overview

The U. S. Department of Defense (DoD) defines a simulation model as a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process, and a simulation as a method for implementing a model over time [Def94]. The DoD also classifies computer simulations in three broad categories called *live*, *virtual*, and *constructive* simulation [US95b]. However, DoD recognizes that the categorization of simulation into live, virtual, and constructive is not entirely definitive because there can be unclear divisions between these categories. The degree of human participation in the simulation is infinitely variable, as is the degree of equipment realism. This traditional categorization of simulations also suffers by ex-

cluding a category for simulated people working real equipment (e.g., smart vehicles) [US98].

According to [US95b] and [US98], each category is defined as follows:

a. Live Simulation: A simulation involving real people operating real systems.

b. Virtual Simulation: A simulation involving real people operating simulated systems. Virtual simulations inject Human-In-The-Loop (HITL) in a central role by exercising motor control skills (e.g., flying an airplane), decision skills (e.g., committing fire control resources to action), or communication skills (e.g., as members of a C4I team).

c. Constructive Model or Simulation: Models and simulations that involve simulated people operating simulated systems. Real people stimulate (make inputs) to such simulations, but are not involved in determining the outcomes.

Embedded Simulation (ES) integrates simulation technology with real systems, providing the operator with a chance to rehearse a mission in the real vehicle, interacting with the virtual world as if it were real, and enhancing training locally and in remote locations. The virtual interaction includes mission rehearsal, battlefield visualization, command coordination, and training.

Objective Force Embedded Training (OFET) methods offer several distinct advantages for 21st century training environments. Benefits include the ability to perform *in-situ* exercises on actual equipment, more direct provision of support for the variety of equipment in the field, and a greater opportunity to develop new training exercises using much shorter lead times than were previously possible with stand-alone training systems [BAC97]. A fully operational OFET platform also presents several technology challenges. In particular, management of Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) resources is required for successful integration of simulation within

the actual environment. This leads to the general project proposal from which this dissertation was motivated as one of the research branches in [GDD02, VGD03].

As pointed out by McDonald [McD88], McDonald and Rullo [MR90], and McDonald and Bahr [MB98a], [MB98b], in the late 1980's Embedded Training (ET) started as an important initiative of the U. S. Army for training army personnel. Some of the reasons for developing ET include budget cuts, security interests, need to train forces by practicing missions without physically disturbing cultural and environmental issues, etc. Other initiatives developed included Embedded Operations (EO) and Embedded Simulation (ES). The three initiatives had areas in common that facilitated the migration among them. The Inter-Vehicle Embedded Simulation Technology (INVEST) program was proposed by the U. S. Army Simulation, Training and Instrumentation Command (STRICOM) office in order to explore fundamental technologies to apply ET and ES to future ground combat vehicles.

Computer Generated Forces (CGF) was a related project sponsored by DOD in the 1990's. The idea behind CGF is that the trainees need opposing forces against which to rehearse, although they can also use them as friendly forces to operate along with [HGG00]. These forces are generated by one or more of the participating sites in the synthetic battlefield. Under CGF the two major efforts were Modular Semi-Automated Forces (ModSAF) and Close Combat Tactical Trainer Semi-Automated Forces (CCTT-SAF). In 1998, STRICOM started to develop a recommendation of the SAF system to be used as the baseline for integrating ModSAF and CCTT-SAF into a OneSAF Testbed Baseline (OTB). OTB was planned to be used for supporting research and development for the next generation of architecture experiments, extending toward providing a Battalion Battle Simulation (BBS) replacement capability through a Battalion Level Behavior (BLB) Application Program Interface (API), and providing the training capacity of CCTT-SAF. Detailed information about the historic development of OTB can be found in [Cor98] and [MWH01].

1.2 Distributed Simulation Environments

As Roger Herdman explains in [Tec95], Distributed Interactive Simulation (DIS) is the linking of several military simulators like tank and aircraft in locations that can be geographically distributed throughout LANs and WANs in such a way that the crew of a given simulator can interact with crews in the other simulators for playing the roles of friendly or opposing forces. The participants can cooperate with friendly forces, and shoot and destroy enemy ones. Command structures are also simulated. In this way, the participants get trained in a broad range of scenarios without risking their lives and at a fraction of the cost of a real operation.

The objective of DIS is to develop standards that provide guidelines for interoperability in military simulations. DIS is a protocol initially specified in ANSI/IEEE Std 1278-1993 Standard for Information Technology, Protocols for Distributed Interactive Simulation [IEE93]. The standard has been refined and extended in [IEE95a], [IEE95b], [IEE96], and [IEE98]. A main contribution of the DIS standards was the definition of the Protocol Data Unit (PDU).

Because DIS is a stateless system that does not utilize servers, reliable multicast communication is used to transmit information like terrain and environmental updates. A Log-Based Receiver-reliable Multicast (LBRM) communication was proposed in [HSC95] as a means to provide efficient DIS communications in high-performance simulation applications. This reliability is given by a logging server that logs all transmitted packets from the source. If a packet is lost, the corresponding receiver asks the logging server to retransmit it. Another important fact of the logging server is that at the end of the simulation the logged PDUs are available for subsequent analysis, which is the case of the OTB logger. One successful DIS application (precursor of OTB) was Modular Semi-Automated Forces (Mod-

SAF), that simulates the hierarchy of military units and their associated behaviors, combat vehicles, and weapons systems [COM96].

A drawback in DIS is its high network bandwidth requirements and the large computational loads placed on host computers. To overcome the problem, an agent based architecture together with smart networks was proposed in [SZB96]. Mobile agents consist of program scripts that are sent over the network to a remote server. They contain state information and the executable code to be run in the remote server, using the Remote Programming (RP) Paradigm. Remote programming is different form the traditional Remote Procedure Call (RPC) in the sense that not only the parameters but also the corresponding procedure is sent over the network. The mobile agent can start its execution in one server, and continue in another one by saving and attaching its state to itself. According to [SZB96], Entity State PDUs (ESPDUs) account for up to 70 % of the network traffic. They are used to communicate any change of state from one entity to the others, once a given threshold is achieved. Also, DIS indicates that entities must send a *heartbeat* message at specified time intervals, usually every five seconds, broadcasting their state, so that if a new entity joins the simulation, it can be informed about all the other entities already present. Also, every simulator broadcasts a *Simulator Present PDU* every 20 seconds as a heartbeat message required by the Persistent Object (PO) protocol implemented in ModSAF and OTB [Kir95]. If an entity is moving, ESPDUs are sent at a higher rate than if it is still, but even still entities have to inform its position at a given rate. Mobile agents can lower the usage of ESPDUs by maintaining the positions of the still entities, instead of constantly sending ESPDU messages.

1.3 Need for Communication Optimizations

In an embedded simulation system, the participating entities of a mission can be physically separated by long distances, possibly onboard mobile vehicles, and communicating via wireless channels. All the vehicles share a common virtual world that has to be constantly updated, which carries realtime constraints on the bandwidth, latency and connectivity of the subjacent network. OTB, for instance, communicates through the PDU messages under the DIS protocol. Every time an event occurs in a participating entity, like acceleration, firing, detonation, etc., a PDU is broadcasted, making all the other entities aware of that event. Even if nothing special is happening, the entities generate an ESPDU every five seconds as a heartbeat to inform that the entity is still up and running [SZB96, Sri96].

In distributed simulation exercises it has been found that 50 % to 80 % of the network traffic is originated from updates transmitted to ensure that all the simulators have consistent information about the entities participating in the simulated battlefield [CD96]. In order for the participants of the simulation to interact with the virtual world in a realistic way, they must see and communicate with each other in real time. To accomplish this, each simulator maintains *dead-reckoning* models of its own state and of the state of all other vehicles with which it may interact, and so the network used for embedded training must be able to transfer massive volumes of data [HGG01].

Scalability is not only desirable, but a requirement of current simulation protocols like HLA [WJ98]. Bandwidth is a scare resource, and the larger the number of participating sites, the more compromise the available bandwidth becomes. Stone [SZB96] indicates that the greatest problem currently facing the progress of distributed simulations is scalability, and that it is very difficult to scale up beyond

approximately 2,000 entities due to the tremendous requirements for network bandwidth.

Several attempts have been made to overcome the bandwidth problem. The general idea relies on finding new methods or algorithms to reduce the network traffic, either by applying some lossless compression algorithms, by eliminating some redundant packets, by splitting some PDUs into static and dynamic data and sending the static data once and the dynamic data more often (delta-PDUs), by concatenating (bundling) some PDUs into a larger packet that is later split at the destinations into individual PDUs (replication), by re-scheduling some PDUs from high intensive traffic spikes to periods of lower traffic demands, by using multicasting instead of broadcasting, by applying priorities to PDUs and using Head-of-Line (HoL) algorithms at router queues, or by applying a mix of all these ideas.

In this dissertation, some of the previous methods are investigated and enhanced. Re-scheduling of the PDUs attempts to alleviate the occurrence of spikes of negative slack time when OTB timestamps PDUs at exactly the same time. The basic idea is that it is possible to slightly modify those timestamps in such a way that the overall simulation is not affected, while exploiting the time interval of positive slack occurrences following the negative ones. The re-scheduling effect is automatically achieved by bundling those PDUs and sending a single packet at a slightly later time.

Bundling and replication deal with sequences of several consecutive PDUs timestamped at the same time or almost the same time, for instance PDUs of type *po_fire_parameters*, which are the main cause of the said negative spikes. Basically, these PDUs are copies of an initial reference PDU. Then, a new method that eliminates all the duplications is proposed. Bundling occurs at the sending sites and replication is performed at the receiving ones.

1.4 Outline of Dissertation

The remainder of the document is divided into the following Chapters. In Chapter 2, *PREVIOUS WORK*, a review of the State of the Art in bandwidth assessment for Embedded Simulations is given. The section *Bundling And Aggregation of Network Packets* deals with current techniques for bundling PDUs. The section *Data Compression Techniques* mentions some common compression techniques belonging to the *loss* and *lossless* categories. The section *Data Transmission Optimizations* refers to the possibility of PDU rescheduling as a means of diminishing high traffic demands over brief intervals of time. The chapter ends with the section *Comparison of Bundling Strategies* that contrasts the investigated techniques to make the most of the available bandwidth.

In Chapter 2.4, *COMMUNICATION RESOURCES AND ARCHITECTURE*, the Flying LAN is presented and serves as a framework for the communication resource model. The OMNeT environment is introduced along with the key concepts of model design, primitive and compound modules, and instantiation of the network applied to the simulations at hand. The Chapter ends with a review of some theoretical aspects of Head-of-Line priority service in queues.

In Chapter 4, *PACKET ALLOYING BUNDLING TECHNIQUE*, the concepts of offline and online bundling are stated, and how they relate to the algorithms proposed in this dissertation. The characteristics of embedded simulation traffic impacting bundling are exposed, the mathematical description of the *Packet Alloying* bundling strategy is given, and a description of the proposed offline and online algorithms is provided.

In Chapter 5, *EMBEDDED SIMULATION TRAFFIC ANALYSIS*, four experiments are described. The general format of the input data is explained, and two examples of actual PDUs are given. Simulation results are graphically presented

for each one of the experiments. In each case, an independent analysis consisting of bandwidth statistics calculated before running the simulation are shown as a means of predicting and corroborating the simulation outcomes. The PDU traffic is then analyzed considering the criteria of slack time, travel time, queue length and collision analysis. Spike analysis resulting from many observed negative spikes in the slack time of the senders is studied in one of the experiments. A sample of some negative spikes is collected, and the corresponding PDUs are identified, resulting in interesting observations about the constant appearance of some PDUs like *po_fire_parameters*. These observations determine fundamental parameters for the proposed algorithm called *Packet Alloying*.

In Chapter 5.8, *TRAFFIC OPTIMIZATION USING PACKET ALLOYING*, the proposed bundling algorithm *Packet Alloying* is tested and its behavior is analyzed by creating custom models and simulating them. Comparisons against the non-bundling approach are given. The conclusions indicate that Packet Alloying is an effective algorithm, with significantly improved performance over its non-bundling counterpart.

In Chapter 7, *CONCLUSION*, the results of the experiments are summarized and general conclusions about the simulation tool, the methodology employed, PDU traffic and minimum bandwidth requirements, are drawn. Future work is identified for the continuation of the project. Several areas are proposed for further research, related to new bundling options, better prediction tools for upcoming PDUs, and the application of these techniques to other communication protocols.

1.5 Contributions of Dissertation

A summary of the main contributions made by this dissertation areas follows:

1. **Offline Analysis:** The formalization of the independent offline analysis for packet traffic based on availability of logged PDUs aimed at the assessment of minimum bandwidth requirements for the network. The independent analysis provides a first approximation to the minimum bandwidth requirements, which is more computationally tractable and more direct than performing a packet-based discrete event simulation, perhaps having to run it several times under different parameter combinations. Applied to the studied vignette, the independent analysis estimated the required bandwidth on the order of 200 Kbps, a value that was later confirmed by the OMNeT model simulation.

2. **Packet Alloying:** The formalization of a selective packet bundling strategy, called *Packet Alloying*. This bundling technique becomes more active when it is needed the most: during negative spikes of the slack time, and produces new packets that preserve the internal PDU format. The resulting packets can be considered as properly preserving the formats of PDUs. Due to that characteristic, bundled PDUs are subjectable to further compression by conventional techniques. For example, if A and B are PDUs such that $A = (a_1, a_2, a_3, a_4)$, $B = (b_1, b_2, b_3, b_4)$, and $a_2 = b_2$, $a_4 = b_4$, then the bundle $A \otimes B$ is represented as $(a_1, a_2, a_3, a_4, ((b_1, 1), (b_3, 3)))$. For instance: $(10, 8, 12, 20, 9) \otimes (10, 6, 12, 20, 3) = (10, 8, 12, 20, 9, ((6, 2), (3, 5)))$. Packet Alloying proved to be useful for bundling *po_fire_parameters* and other high priority PDUs.

3. **Sequence Prediction:** The proposal and study of different predictive algorithms for the next PDU, three of them online: *Neural-Network*, *Always-Wait* and *Always-Send*, and three offline: *Type*, *Type-Length* and *Type-Length-Size*. Applying *Always-Wait* to the studied vignette and setting the wireless links to 64 Kbps, a reduction in the magnitude of negative slack time from -75 to -9 seconds for the worst spike was achieved, which represents a reduction of

88 % over non-predictive transmission. Similarly, at 64 Kbps *Always-Wait* reduced the average satellite queue length from 2,963 to 327 messages for a 89 % reduction. These performance improvements are quite significant and allow vignettes to be simulated using communication channels with bandwidths as low as 64 Kbps.

4. **Priority-based Optimization**: Use of Head-of-Line (HoL) priority service in router and satellite queues based on PDU type. From the analysis of negative slack-time spikes, it was determined which PDU types are of highest priority. Next, a priority-based transmission scheduler was developed. The analysis of total travel times based of PDU types numerically shows the benefit obtained. For example, at 128 Kbps, it is seen that by incrementing the delay of low priority ESPDUs by 6.8 %, produces a decrement of 9.8 % in the latency of *po_fire_parameters*, *po_line* and *po_task* PDUs, 5.6 % in the latency of *po_task_state*, and 1.9 %, in all other PDU latencies.

The results of applying the said techniques demonstrate the validity of these packet optimizations for assessing bandwidth in distributed simulation environments. These techniques also can be readily extended to other possible applications such as TCP/IP transmissions.

CHAPTER 2

PREVIOUS WORK

Many different solutions aimed at decreasing network traffic have been studied in the literature, including bundling, delta-transmissions, dead-reckoning, relevance filtering, compression, multicasting, quiescent entities, and the use of unreliable transport mechanisms. Some of the most common and relevant bundling-related mechanisms are explained in this Chapter.

2.1 Bundling and Aggregation of Network Packets

Several authors have contributed to the principle of bundling packets, not only applied to the DIS protocol, but also in other fields. Back in 1988, Baum and McMillan applied the concept to messages traversing an hypercube network. They investigated a mechanism for reducing the communication cost by bundling together messages sent along the same channel, and concluded that the additional overhead required to bundle the messages at the sending processor and to unbundle them at intermediate processors is not large [BM88] and can be beneficial at elevating effective bandwidth at the application level.

More recently, Calvin and Van Hook have proposed very similar definitions of bundling. They say that bundling combines PDUs into larger packets in order to reduce packet rates. A packet is transmitted when either a timer expires or the

packet reaches a maximum size. As a consequence, the necessary bit rates are reduced since fewer packet headers are transmitted, placing multiple DIS PDUs in one single packet for transport, obtaining improvement results [CST95, VCR96]. Each packet header is 12 bytes long [IEE95a], but including the Ethernet header and encapsulation information, the packet header grows up to 42 bytes [WJ98]. Fewer packet headers imply also fewer inter-packet gaps that contribute to bandwidth savings.

Frederiksen and Larsen furthered the bundling concept by introducing a new parameter to the bundling discussion: the necessary gap that must exist to separate physical packets in a communication channel. They say that if data to be sent becomes available a little at a time at irregular intervals, the sending side must decide whether to send a given piece of data immediately or to wait for the next data to become available, such that they can be sent together as a bundle [FL02]. The decision of sending is not trivial because of physical properties of the networks requiring that after sending each packet, a certain minimum amount of time or *gap* must elapse before the next packet may be sent. Thus, whereas waiting for more data will certainly delay the transmission of the current data, sending immediately may delay the transmission of the next data to become available even further [FL02].

In a recent article related to DIS applications, Ceranowicz describes the Joint Experimental Federation (JEF) and the Millennium Challenge 2002 (MC02), a simulation conducted in July and August of 2002 by 13,500 personnel at locations across the United States. In the article, he reports about the maximum limit of bytes that can be bundled. In one of the experiments, up to to 4,500 bytes were bundled in each IP packet and updates were collected for up to one second. He concludes that the tradeoffs were that bundling more data together increased latency and packet loss due to transmission errors, while smaller packets increased the transmission of overhead data [CTH02]. This shows the importance for choosing the proper bun-

dle size and we have address this impact in this dissertation. More formally, these important parameters have been defined in terms of the transmission rate available.

In [BCL97] and [LCL99] consecutive PDUs are *concatenated* in a single packet even if their types are different, and redundancy in the fields that make up a PDU is not eliminated. Bassiouni explains that the benefit of PDU bundling comes from the fact that network routers, bridges, gateways, and computer hosts have a limited bound on the number of packets that they can process or transmit per second, and bundling can effectively increase this bound. Bassiouni and Liang have the same formalization of PDU bundling, which follows:

> Let r_s denote the maximum number of PDUs of size S bytes that can be transmitted from host A to host B in one second. Suppose that host A starts bundling its transmitted PDUs, instead of sending them as individual packets, by assembling k PDUs into a larger packet that is transmitted as a single unit. Let r_{ks} be the maximum number of packets per second that can be transmitted from host A to host B if k PDUs are bundled into a single packet, where $k > 1$. In many networks under most loading conditions, the following relationship holds [BCL97, LCL99]:
>
> $$r_s < k \cdot r_{ks} \qquad (2.1)$$
>
> and the percentage gain in the PDU peak rate is $100(k \cdot r_{ks} - r_s)/r_s$.

The product ks in inequality 2.1 implies that the proposed bundling mechanism does not compress or reduce the size of the bundled PDUs. The PDUs are just concatenated regardless of their internal structure, type or redundancy. It is also interesting to note that Bassiouni and Liang indicate that some time-critical PDUs like *fire*, *detonation* or *explosion* cannot be effectively bundled. In this dissertation

those types of PDUs are bundled, given that they are scheduled at the same time and are subject to an unavoidable delay caused by the satellite link, facts that make the incurred bundle delay negligible.

Liang also proposes bundling using *Multilevel Priority Queues* (MPQ) as well as *Single Priority Queues* (SPQ) [LCL99]. Both mechanisms are variants of the *Head-of-Line* (HoL) strategy from queue theory referenced by many authors, for instance [LS93, DGR01, Liu02, PW03, GM04]. The general idea in HoL algorithms is to assign a priority to the incoming elements (PDUs, cells, frames, etc.) and using a priority queue, serve the higher priority elements first, possibly defining timeouts for the low priority ones so that starvation is prevented.

A *Delta-PDU Encoding* technique is mentioned in [US95a, Mac95] consisting of PDUs that carry changes respect to a reference PDU that is provided initially. The technique exploits the fact that most information in DIS entity state PDUs is redundant from one packet to the next. The delta-PDU encoding is accomplished by splitting the DIS Entity State PDUs into *static* and *dynamic* data PDUs. The static data becomes the reference PDU, while the dynamic one carries the changes. The idea has also been studied for the HTTP protocol of the internet using intermediate proxy servers as cache memory [MDF97, MDF02, WAS96]. Wills [WMS01] describes several delta encoding and bundling techniques generally applicable to Web pages under the TCP/IP protocol suite, but none is specific to the DIS protocol.

A protocol called *DIS-Lite* developed by MäK Technologies [Tay95, Tay96b, Tay96a, PW98] also splits the Entity State PDU into static and dynamic data PDUs, so that the static information is sent once and the changes (dynamic PDUs) are subsequently sent as separate PDUs. DIS-Lite offers several advanced features, including packet bundling, latency compensation and enhanced dead-reckoning algorithms tailored for air vehicles [PW98]. According to Fullford [Ful96], by eliminating redundancy DIS-Lite can perform between 30 % and 70 % more efficiently

than DIS. DIS-Lite also includes several other improvements not related to the combination of individual fields from a set of similar PDUs. These objectives complement related predictive strategies developed for conserving simulation bandwidth [BD96, HGG01]. Finally, the bundling principle is also applied to *Time Division Multiplexed* (TDM) networks, where a number of lower order frames are multiplexed into one higher order frame [Ber02]. These all indicate the potential benefit of packet aggregation, but a specific strategy that can operate in real time for embedded simulation environments have not been provided, nor have their performance been assessed to determine an optimal lightweight strategy for DIS.

2.2 Data Compression Techniques

Data compression has been long studied and many papers have been written on the topic. One of the goals of this dissertation is to diminish the bandwidth requirements of OTB simulations by reducing the PDU content transmitted over the network. It has been observed that in many cases, subsequent PDUs are almost identical to previously transmitted ones. This observation leads to the conclusion that it is possible to apply one or several compression techniques to achieve better bandwidth utilization.

Generally speaking, compression algorithms can be classified in two broad categories: *lossy* and *lossless* algorithms. Lossy compression corresponds to algorithms that do not guarantee an exact recovery of the compressed data. In many applications like sound and video this loss is acceptable because human senses do not detect the faults in the uncompress data, or because the final quality of the uncompressed data is acceptable.

Lossless compression involves algorithms that can recover the original data without faults. Applications include the transmission of an executable binary file, or the compression/uncompression of TCP/IP packets. In this research, the lossless compression of PDUs is sought. A detailed treatment of loss and lossless compression algorithms is found in Deorowicz's PhD dissertation [Deo03].

Compression algorithms can also be classified by the method employed to compress the data. Some methods are based on dictionaries, guess tables or a mix of both [Hew95, TS02]. Methods based on dictionaries create a dictionary of strings commonly repeated in the data and corresponding keys much shorter than the strings. Then, instead of the string, the key is transmitted. Obviously, each communicating party needs to know the dictionary, which is transmitted in advance. One of the most common lossless compression algorithms based on dictionaries is the Lempel-Ziv (LZ) algorithm [ZL77]. Common compressors like *gzip*, *winzip* and *pkzip* are based on the LZ algorithm. Guess tables are based on the idea that certain bytes can be guessed from the previous transmitted ones. Both, the sending and receiving sites maintain the same guess table. If the transmitter can guess the next byte, then that byte is not transmitted but entered into the table.

Some algorithms are based on the concept of entropy taken from the information theory to produce high compression rates. They are related to Shannon's fundamental source coding theorem [Sha48] and an example of this kind of algorithms is Huffman coding [Huf52]. In [FY94] a lossless algorithm to compress volume data based on differential pulse-code modulation (DPCM) and Huffman coding is presented.

Compression algorithms have been devised specifically to compress network packets. Dorward [DQ00] presents such an algorithm based on a variant of Lempel-Ziv's compression, and Ishac and Degermark [Ish01, DNP99] deal specifically with TCP/IP headers. If the data to be transmitted in the TCP/IP packets is too small,

the transmission overhead of the headers starts to be an important factor. The header compression combined with data concatenation of several small PDU packets is then an appealing technique. According to Degermark, header compression can decrease the header overhead for IPv6/TCP from 19.5 % to less than 1 % for packets of 512 bytes.

TCP/IP compression has been researched by companies like Dataline (`http://www.dataline.com/`) that claims that by using its TCP/IP acceleration technology Joint En-route Mission Planning and Rehearsal System - Near Term (JEMPRS-NT) can deliver the functionality required by a traveling team of 12-15 users over a satellite link. Dataline affirms that by using its technology, a 64 Kbps link can give the equivalent throughput of a virtual 256 Kbps [Fro02]. As an interesting observation, the simulations run in Chapter 5.8 using the proposed bundling algorithm Packet Alloying, produced results comparable to Dataline's affirmation, just by employing bundling alone. The cascaded usage of several compression strategies could lead to even better results.

There are two general classes of compression techniques for removing PDU redundancies: *application dependent* and *application independent* strategies. Algorithms that fall in the first group know and take advantage of characteristics of the simulation application like encoding data based on bit strings typical of the application and sending delta PDUs. In the second group, algorithms are more general, do not know particularities of the application and work by examining and compressing the bit strings by detecting bit patterns. According to Van Hook [VCN94], application dependent techniques can achieve slightly more compression than the independent counterparts, but are a lot more complex and require much more processing power.

In 1994, the Advanced Research Projects Agency (ARPA) simulation exercise called Synthetic Theater of War-Europe Application Gateway (STOW-E AG) was conducted to test a new communication architecture. The AG was installed be-

tween each site LAN and the WAN to apply several algorithms and techniques for managing traffic flowing to and from each site, like blocking unnecessary PDUs, Protocol Independent Compression Algorithm (PICA), grid filtering, Quiescent Entity Service (QES), rethresholding, bundling, load leveling. Calvin reports that the application of all these techniques produced a reduction in network traffic by more than an order of magnitude [CST95].

Similarly, *PICA* was originally proposed as a compression algorithm by Van Hook in 1994 to compress SIMNET PDUs [VCM94], achieving up to 76 % reduction in bit rate. The SIMNET protocol and simulation was a precursor of DIS protocol developed in the late 1980's by the Defense Advanced Research Projects Agency (DARPA). PICA compresses ESPDUs by transmitting a reference ESPDU that becomes known to the communicating entities, and subsequently sending delta-PDUs, also called Differential Data Units (DDUs), containing those bytes which differ from the reference ESPDU. Eventually, a new reference PDU, called Differential Key Data Unit (DKDU), is transmitted because at that time the compression being achieved by PICA falls below a threshold, due to increasingly larger bit pattern differences [DCV94, VCN94, Fuj95]. PICA has been reported to yield fourfold compression of entity state PDUs, although Fire, Detonation, and Collision PDUs are not compressed before bundling, due to their relatively few number and small size [VCN94]. However, in this dissertation, `po_fire` PDUs are successfully compressed because of the conditions (similar timestamps, long satellite link delays) specified for the vignette.

2.3 Data Transmission Optimizations

One of the goals in this dissertation consists of rescheduling some of the PDU packets in order to better utilize the bandwidth by transferring PDUs from periods of high activity to periods of low activity. As stated in the introduction, one of the main causes of negative slack spikes is the scheduling of several PDUs at the same time, which causes a bottleneck in the transmitting sites.

Packet rescheduling is an old technique, initially developed for the Ethernet CSMA/CD protocol 802.3 [IEE85] to manage collisions during the contention period. The *exponential backoff* algorithm is commonly used to reschedule the collided packets, although Molle considers that the stability of the algorithm is somewhat open to debate [Mol94]. If PDUs are timestamped at the same time, they can be considered as a kind of collision that needs to be solved. The scheduling of packet networks at the router level has been recently studied in [APR03] where a randomized parallel scheduling algorithm for scheduling packets in routers based on the Switch-Memory-Switch (SMS) architecture is developed.

Multicast routing algorithms and protocols with emphasis on QoS is addressed in [WH00]. The network is represented as a weighted digraph with one or more parameters associated to the links. Each parameter represents a characteristic of the link, like transmission and propagation delay, bandwidth, etc. The nodes also contain parameters that describe their stauts, like buffer space available, queue length, etc. The multicast routing problem consists of finding minimum spanning trees for a given objective function subject to QoS constraints. The problem is classified into several categories depending on the objective functions to be minimized and the QoS constraints. Examples of those categories are: link constrained problems, tree constrained problems, link and tree constrained problems, tree optimization problem, etc.

Packet transmission strategies using priorities have been widely studied. The *Head-of-Line* algorithms and priority queues are examples that make use of packet priorities. A method that exploits a priority scheme to guarantee static preplanned message slots for hard real-time communication is found in [KLJ00]. The mechanism is embedded in the MAC layer. The method considers that there are highly time-critical messages that require bounded transmission times. Applied to the present simulation, these time-critical messages could be the *po_fire_parameters* PDUs involved in the negative slack spikes. Priorities are also used in real-time applications when the traffic is shared with non real-time ones. A recent PhD dissertation by Pope [Pop02] addresses the issue of bounding packet delay based on various queuing disciplines under real-time constraints, and the results presented in this dissertation proceed further along these lines.

2.4 Comparison of Bundling Strategies

Table 1 summarizes the most common bundling-related strategies used to conserve bandwidth over WAN networks.

Table 1: Comparison of Bundling Techniques

Technique	**Advantages**	**Limitations**
Concatenation	Reduced headers and ACKs, fewer gaps, fewer collisions	Limited to maximum size of frames, redundancy not eliminated, not used with fire and detonation PDUs
IP header compression	Savings in header transmissions	Applicable to TCP/IP, not PDU packets, good for small packets, low compression ratio, redundancy not eliminated
Delta-PDU	Advantages of concatenation, high compression ratio	Dependency on a reference packet
DIS-Lite	Advantages of delta-PDU, includes other compression techniques	Disadvantages of delta-PDU, tailored for air vehicle simulations

The first technique is plain concatenation of PDUs. Consecutive PDUs are concatenated so that the total length of the bundle is equal to the sum of the components. In this process, PDU types and timestamps are not considered in the decision of concatenating PDUs. The technique is usually applied to ESPDUs, but not to fire and detonation ones. During concatenation, each PDU conserves its own PDU header and data fields. As advantages of concatenation, we could point out the savings in packet headers and ACK replies at the transport layer, and fewer separator gaps at the physical layer, as a result of using a single header, a single ACK and one gap instead of many of them. Also, fewer collisions are produced by having to acquire the channel fewer number of times, assuming a CSMA/CD link. Among the disadvantages, concatenation should be limited to the maximum size of a network packet, to keep the previous advantages. Redundancy is not eliminated, and so fewer packets can be bundled in the same block, as compared to a technique that eliminates it.

The second technique is IP header compression, in which consecutive IP headers having high similarity are compressed by a lossless algorithm. As will be shown later, it is generally applicable to TCP/IP packets, but not to PDUs. The impact is enhanced if data segments are small in the packet, because then the IP headers are proportionally larger. Advantages include transmission of less data due to a shortening of headers. Disadvantages are a low compression ratio for PDUs, given that their data fields are not compressed and are usually long. Additionally, it does not eliminate redundancy.

The third technique is based on the usage of Delta-PDUs. Here a reference PDU is transmitted first, followed by packets carrying the differences only. Destinations must keep the reference PDU in order to recover the original PDUs from the deltas. When the delta PDUs have reached a discrepancy larger than a threshold, a new reference PDU is transmitted. Advantages include all the advantages of plain con-

catenation, plus a high compression ratio. The delta PDUs are not constrained by the network packet size, because each delta is sent in a separate packet. A disadvantage is that delta PDUs are dependent on the reference PDU. If the reference PDU gets lost or out of sequence, assuming unreliable UDP transport, the deltas become useless and the entire sequence has to be retransmitted.

The fourth technique is called the DIS-Lite protocol. It implements delta-PDUs and many other optimizations not related to bundling. DIS-Lite terminology calls the reference and delta PDUs *static* and *dynamic*, respectively. It was originally proposed for air vehicle simulations, but its concept has been applied other areas like video games played on the internet. Its advantages include all the advantages of delta-PDUs, as this technique is a refinement of delta-PDUs, but has been improved by using several other compression and bandwidth-saving techniques. The technique has been applied mainly to the compression of ESPDUs, which is a disadvantage if an environment accepts compression of other PDUs. Also, it does not examine and take advantage of the type, timestamp and internal structure of PDUs. It is mainly targeted at simulations of aircraft vehicles.

In this dissertation, enhanced techniques beyond concatenation extend the previous work in a specific direction. The focus is on lossless strategies for packet aggregation within the application area of distributed simulation. Advantages of IP header reduction, delta PDUs, and concatenation are combined.

CHAPTER 3

COMMUNICATION RESOURCES AND ARCHITECTURE

Several communication case studies from the OneSAF Testbed Baseline (OTB) are assessed for multiple-platoon, company, and battalion-scale force-on-force vignettes consistent with Future Combat Systems (FCS) Operations and Organizations (O&O) scenarios. Traffic is modeled using OMNeT++ discrete event simulator models and scripts developed for a hierarchical communication architecture consisting of eight enroute C-17 aircraft each carrying three Ethernet-connected M1A2 ground vehicles, a wireless flying LAN based on Joint Forces Command's Joint Enroute Mission Planning and Rehearsal System (JEMPRS) for Near-Term (JEMPRS-NT) and follow-on bandwidth capacities. The simulation model is presented in detail, including the OMNeT characteristics necessary to understand it. The topology of the network is defined using the NED language and the behavior of each object is defined in C++ code. The simulation traffic includes Opposing Force (OPFOR) control via a CONUS-based ground station and the corresponding satellite links.

3.1 EMPR Simulation Vignette

A simulation environment aimed at assessing One Semi-Automated Force (OneSAF) communications bandwidth during mission rehearsal of Future Combat System (FCS) vignettes was developed at the Electrical and Computer Engineering Department of the University of Central Florida, sponsored by the U.S. Army Program Executive Office for Simulation, Training & Instrumentation (PEO STRI), formerly STRICOM.

Activities were undertaken to understand FCS mission rehearsal operations and define a vignette to generate the simulation traffic to be modeled. The Operational and Organizational (O&O) document [For02] entitled U. S. Army Objective Force Operational and Organizational Plan for Maneuver Unit of Action, TRADOC 525-3-90 O&O dated July 22, 2002 was obtained and reviewed for adaptation to our vignette.

Several different FCS vignettes were prepared and simulated on a Local Area Network (LAN) using the OneSAF Testbed Baseline (OTB) software, a military simulation application that implements a Joint En-route Mission Planning and Rehearsal System (JEMPRS) in an FCS environment. In [VDG04a, VDG04b], the author describes an initial implementation of the OMNeT simulator used, and corresponding results obtained when the vignette logs were run.

Traffic logs were created from the participating sites. OTB communications are based on the Distributed Interactive Simulation (DIS) protocol defined in the IEEE Standards 1278.1 [IEE95a], 1278.2 [IEE95b], 1278.3 [IEE96] and 1278.1a [IEE98]. The fundamental communication packets under DIS are the Protocol Data Units (PDUs), which were logged including relevant PDU information used for alternative parameter variations of the model. The information logged included the type, length, and time-stamp of each PDU.

In particular, the MR1 vignette illustrating a mission rehearsal operation while en-route to deployment and reproduced in Appendix A, was used to generate the PDU traffic logs. This vignette was partly based on and extends TRADOC PAM 525-3-90 Operations & Organizations (O&O) document, "Annex F - Unit of Action Vignettes [For02]." In the section called *Statement Of Required Capabilities For Future Combat Systems* the TRADOC PAM document indicates that the Unit of Action (UA) must be able to integrate into Enroute Mission Planning and Rehearsal Systems (EMPRS) during alert, deployment and employment. FCS and Unit of Action C2 systems must access enroute mission planning, and support mission rehearsal, battle command, and ability to integrate into gaining C2 architectures during movement by air, land and sea. The document contains three vignettes including Entry Operations, Combined Arms Operations for Urban Warfare to Secure Portion of Major Urban Area, and Mounted Formation Conducts Pursuit and Exploration.

The duration of the MR1 vignette is approximately 25 minutes of simulation time. It involves Entry Operations and Maneuver to Attack of a battalion-sized unit tasked with pursuing an enemy delaying force immediately upon landing. The lead elements (Alpha company) detect a fortified position between the main elements and the target enemy force. Four RAH-66 Comanche helicopters are deployed and follow closely. Next, the East friendly forces, begin to advance on the enemy position, however, they must traverse minefields during their pursuit. The enemy force flees southward from the North and East force. The South force engages the enemy, and is assisted by the North and East forces.

The general scenario is that a Battalion Task Force equivalent has been rapidly deployed. There are 8 aircraft, C-17 equivalent, in formation, each one carrying up to three ground vehicles. Inside each aircraft, the vehicles are connected to each other, and also to the aircraft communication resources, via a hardwired Ethernet-

type network. Each ground vehicle contains a computer station running the MR1 vignette on OTB. The aircraft are in communication with each other via satellite that also provides a link to a Continental United States (CONUS) ground station. This ground station provides core exercise support including Semi-Automated Forces (SAF). Additional links are utilized directly between the aircraft to reduce demands on the satellite feed. This is based on SECOMP-1 / JEMPRS Near-Term (JEMPRS-NT) architecture as of January 2003.

Figure 1 shows the model of the flying network used for rehearsal and training on the MR1 vignette. The number of airplanes and simulation stations onboard is variable in the model. The three simulation stations onboard each airplane are connected at 100 Mbps. Connections from plane to plane are achieved via routers and wireless links. Possible values for the wireless bandwidths range from 64 Kbps to 1,024 Kbps. Because the aircraft are flying in formation, the network is not considered an ad hoc network.

The main stages in the development of the model included:

1. Design, obtain approval, and using OTB construct a vignette illustrating mission rehearsal enroute to deployment.

2. Analyze the steady-state and bursty traffic to determine network bandwidth requirements as a proportion of capacity in the tactical C4ISR network.

3. Assess latency and degradation of message delivery due to routing delays, queuing time, and network loading.

4. Improve the network traffic by eliminating redundancy and possibly compacting the PDUs which are the network packets used by the OTB software.

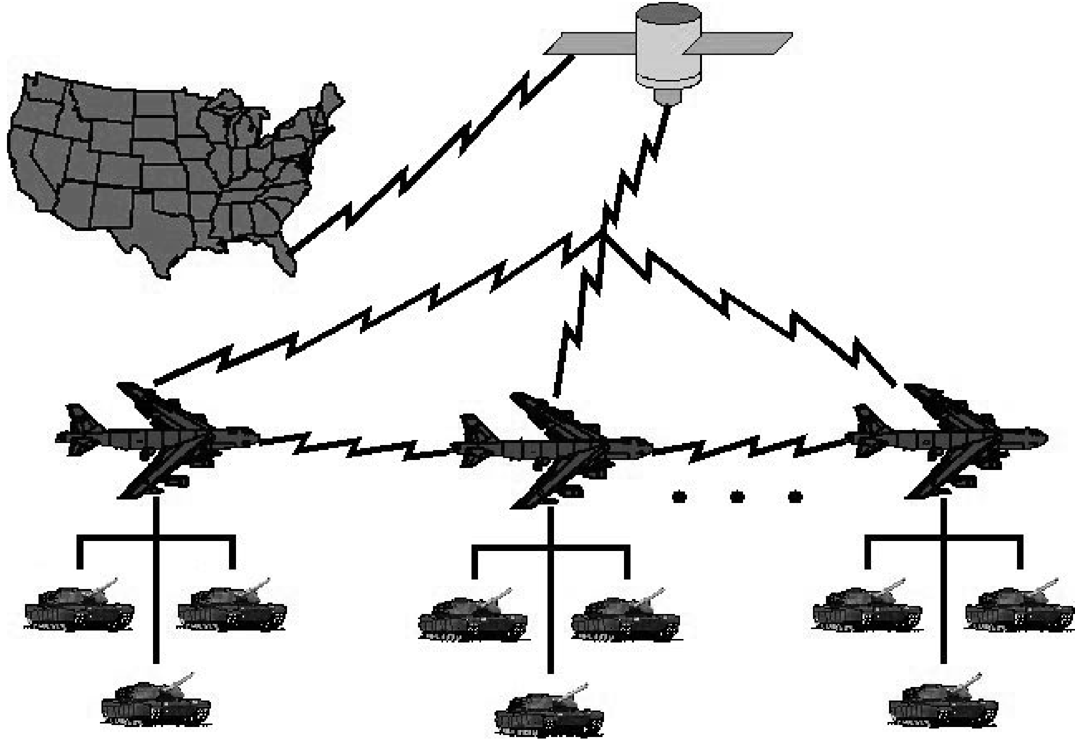


Figure 1: The Flying Network

The purpose of this architecture is to provide a validated application to assess the bandwidth in the wireless links and to develop improved strategies that more effectively utilize the available bandwidth.

3.2 ES Communication Architecture

After evaluating a number of possible configurations, a suitable communication infrastructure was defined and represented in Figure 2, which depicts the communication architecture model used in most of the experiments performed. The simulated model consists of eight airplanes flying in formation towards deployment. Each aircraft carries three ground vehicles, and each vehicle contains a workstation running OTB. Due to the proximity of the aircraft and the fact that they are flying in a

steady formation, all the planes and workstations conform to a non-ad hoc flying LAN wireless network [VDG04a].

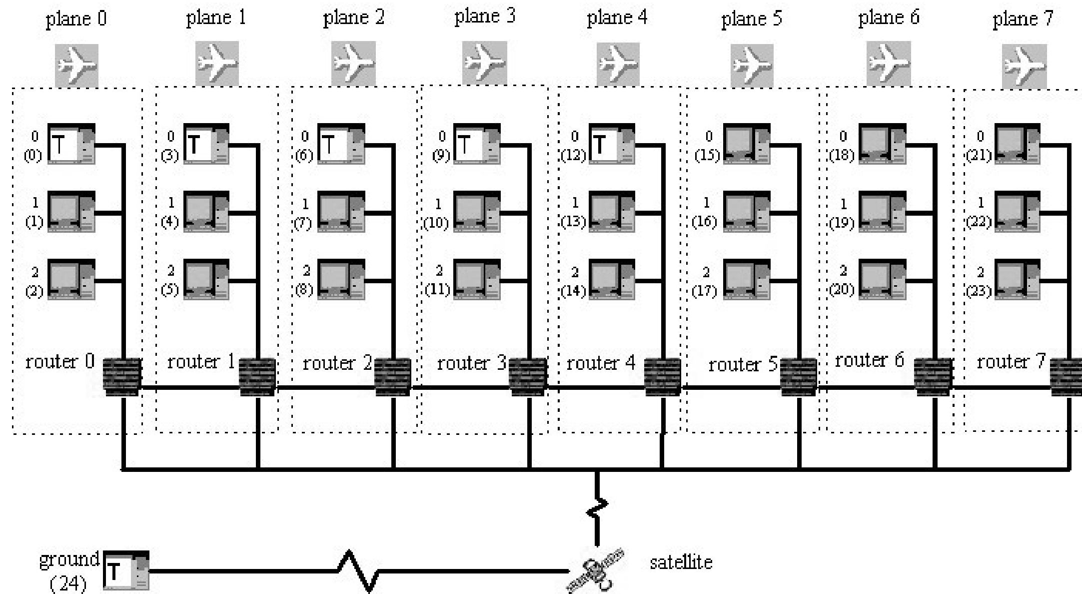


Figure 2: Communication Architecture Model. Sites flagged T are transmitters, the others are receivers.

Planes are numbered 0 to 7, and stations local to each plane are numbered 0 to 2, as well as 0 to 23 for the global network. The CONUS ground station is numbered 24. Routers are numbered the same as the plane they are onboard. According to the data logged form the vignette, some stations are transmitters while others are just receivers. The stations flagged "T" represent packet transmitters; the others are the receivers. Yet, according to the DIS protocol, the transmitters broadcast their packets, and so transmitters are receivers, too.

The number of airplanes, computers onboard and channel bandwidth is not a tight restriction in the model. The three computers onboard the airplanes are connected via Ethernet cable or similar LAN. Connections from plane to plane are provided via routers and wireless links. The bandwidth of the wireless connections is initially set to 64 Kbps, and different runs of the simulator using speeds of 128,

200, 256, 512, and 1,024 Kbps are carried out. The Ethernet LAN is maintained at 100 Mbps in all cases, due to the fact that this technology is common, and the LAN bandwidth is at least two orders of magnitude greater than the wireless bandwidth. A ground station is connected to the flying network through a satellite link. All the computers in the network use the DIS protocol to broadcast messages as specified in [IEE95a].

An object of type *bus* models all the communication links. A bus contains pairs of input and output connectors (IC_i, OC_i) located at known distances d_i from one of its endpoints. Both connectors in the same pair are located at the same distance in the bus. Each bus is configured to operate at a specific bandwidth and propagation delay. When a message enters through one of the input connectors IC_i, the bus delivers it to each of the output connectors OC_j at different times depending on the distance and propagation delay of the medium. The bus was programmed so that signals propagate through it in both directions. If IC_i and OC_j represent input and output connectors located at distances d_i and d_j meters respectively, p is the propagation delay in the bus measured in seconds per meter, b is the bus bandwidth in bits per second (bps), and a message of length n bits arrives into input connector IC_i at time t, then when the message reaches any other output connector OC_j the following measures hold:

$$Distance_traveled = |d_i - d_j| \quad meters \tag{3.1}$$

$$Propagation_Delay = Distance_Traveled \cdot p \quad seconds \tag{3.2}$$

$$Transmission_Time = n/b \quad seconds \tag{3.3}$$

$$\begin{aligned} Start_Time_At_OC_j &= t + Propagation_Delay \\ &= t + |d_i - d_j| \cdot p \quad seconds \end{aligned} \tag{3.4}$$

$$\begin{aligned} End_Time_At_OC_j &= Start_Time_At_OC_j + Transmission_Time \\ &= t + |d_i - d_j| \cdot p + n/b \quad seconds \end{aligned} \tag{3.5}$$

The *Distance_Traveled* is the distance from the input to the output connector in meters. The *Propagation_Delay* is the number of nanoseconds it takes any one bit to travel the distance between the input and output connectors at the speed of p ns/m. The *Transmission_Time* represents the time needed to transfer a packet of n bits to the bus at a rate of b bps. The $Start_Time_At_OC_j$ is the time at which the first fit of the message arrives to OC_j, and the $End_Time_At_OC_j$ is the arrival time of the last bit. The start and end times at OC_j are useful to determine collisions. If a message has a time interval defined by start and end times overlapping any other time interval defined by corresponding start and end times, then a collision occurs.

Generalizing, the model can be described as a collection of computer nodes and routers interconnected by different media at several bandwidths. The transmitters broadcast packets at unspecified rates. In order to generate the packets, it is possible to use a specific probability distribution over time. However, to obtain maximum accuracy, a log of the actual packets generated by OTB, including the PDU type, time-stamp, and packet length was used in place of a random distribution function, providing maximal realism to the results.

3.3 Transmitting and Receiving Devices

The OMNeT++ discrete event simulator was used as the main tool for the model setup. OMNeT++ was designed by András Varga [Var03] at the University of Budapest. The kernel was written in C++, and the user specifies additional modules to program the behavior of the entities in the model. The model design follows a bottom-up approach for modeling the communication architecture. Simple modules are built first and compound modules are built on top of the simple ones. Figure 3.3 gives a general view of the simple and compound modules of the simulator

connected together. It is an actual screenshot of the main OMNeT++ window. If the simulator is executed with the animation option activated, each one of the traveling messages is displayed in this window as a small circle moving across the arrow lines carrying an identification label.

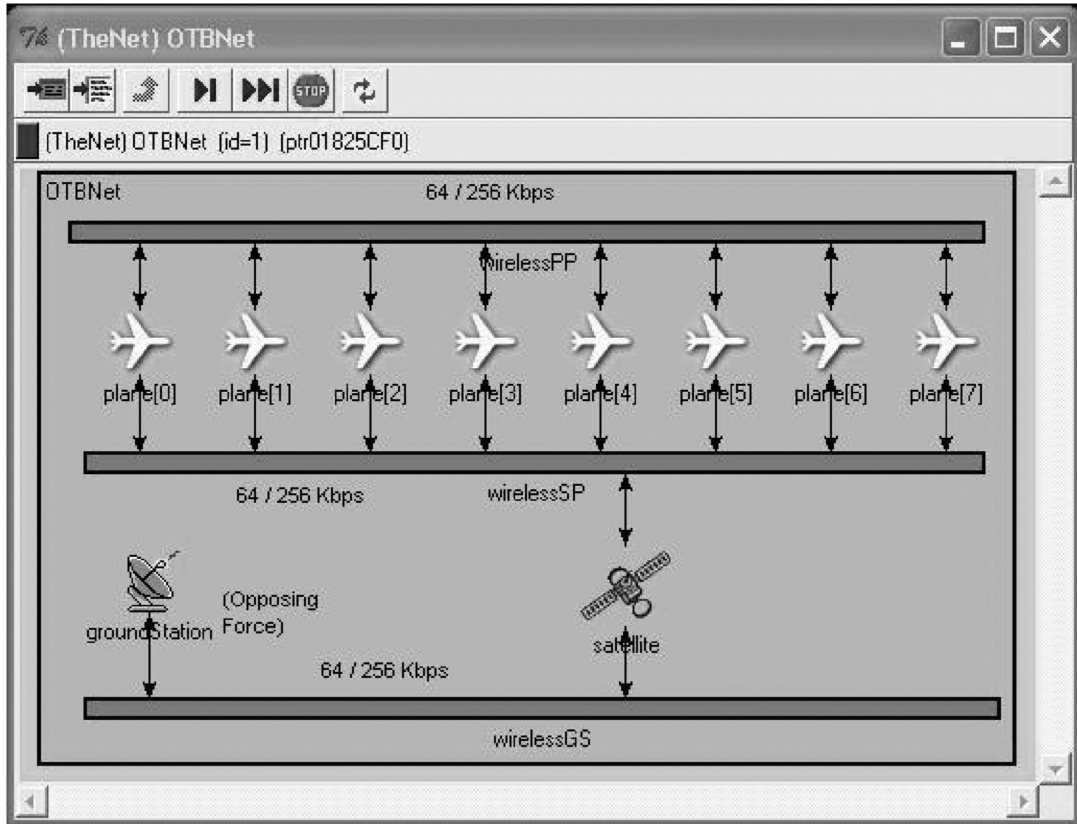


Figure 3: OMNeT Screenshot of the Whole Network Showing 8 Planes, Satellite, Ground Station and 3 Wireless Channels

The entities comprising the model in Figure 3.3 include the four communication links: LAN (not shown), Wireless Plane-to-Plane (WPP, upper horizontal bar), Wireless Satellite-to-Plane (WSP, middle bar), Wireless Ground-station-to-Satellite (WGS, bottom bar), the computer nodes containing a generator and a sink of packets (not shown), routers (not shown), the satellite, and the ground station. Each of the entities was realized as a C++ module.

3.3.1 Primitive Modules in OMNeT

Primitive modules contain no other modules inside them. They are used to describe the most basic elements of the simulator. Generators of messages, sinks or consumers of messages, communication channels (wireless and Ethernet buses), routers and the satellite correspond to simple modules. Each simple module is defined by two files. The first is a *.ned* file that describes input parameters to the module and the set of input and output gates or communication ports. The second is a C++ source file that defines the behavior of the module, i.e. it indicates how to process each message received through any of the input gates and which messages to send through the output gates. The simulator in this project includes the following `.ned` files of simple modules: `generator.ned, simplebus.ned, sink.ned, router.ned`, and `satellite.ned`. Figure 4 through Figure 9 show the corresponding `ned` source codes.

3.3.1.1 The Generators

A generator is the module that produces new messages, following the instructions in the corresponding C++ file. The module reads in a sequence of PDUs from a summary file containing the *type*, *length* and *timestamp*. When the simulation time has reached the `timestamp` of a message, the generator outputs that packet to the LAN link if it is onboard an airplane, or to the WGS link if it is in the ground station. After sending a packet, employing the transmission time corresponding to the packet length and bus bandwidth, an inter-frame space (IFS) or time gap of 50 μ seconds is added, in accordance with the specifications given in ANSI/IEEE protocol 802.11 [IEE99].

The `ned` instructions declaring a generator are given in Figure 4. Due to OTB specifications, all the messages sent by one generator are broadcasted to all of the other nodes.

```
simple Generator       // Generator is a simple module
parameters:
  fromAddr: numeric,   // origin, unique ID within network
  totalNodes: numeric; // number of nodes in the network
                       // (routers not counted)
gates:
  out: out; // The only gate of a generator is called "out"
endsimple
```

Figure 4: Source File Generator.ned

The `fromAddr` parameter is used to give the generator a unique identification. In this simulator, generator IDs range from 0 to 24, where 0, 1, 2 are generators onboard plane 0, 3, 4, 5 onboard plane 1, etc. up to 21, 22, 23 onboard plane 7, and generator 24 is in the ground station.

The `totalNodes` parameter represents the highest ID value assigned to a generator in the model, 24 in this case. The parameter was intended to be used in determining all the valid destination IDs of a message. However, due to the broadcasting feature of the model, the parameter is not actively used in the current version of the generators.

3.3.1.2 The Buses

A `simplebus` is the module that represents the communication links in the network. Instances of it are used to simulate both the Ethernet and the wireless links. Figure 5 shows the corresponding source code of the `ned` file.

```
simple SimpleBus
  parameters:
    busType: string,       // Types: LAN, WPP, WSP, WGS.
    numChannels,           // number of independent channels
    wantCollisionModeling, // collision modeling flag
    wantCollisionSignal,   // "send collision signals" flag
    isFullDuplex,          // channel mode
    delaySecPerMeter,      // delay of the bus
    dataRateBps,           // data rate of the bus
    gapTime;               // minimum gap between packets.
gates:
  in: in[ ];
  out: out[ ];
endsimple
```

Figure 5: File Simplebus.ned

The `busType` parameter indicates the type of link. The possible values of this parameter are LAN, WPP, WSP, and WGS to represent, respectively, the Ethernet link in each airplane, the three wireless links already explained. Each link can be subdivided into several independent channels. The `numChannels` parameter indicates the number of subdivisions. Currently, the simulator is using just one channel per link. The module can be tailored to handle collisions and full/half duplex communications, and the next three parameters indicate this preference. The parameters `delaySecPerMeter`, `dataRateBps` and `gapTime` indicate the propagation delay in seconds per meter, the data rate in bits per second (bps) and the minimum time separation between packets for this link in microseconds, respectively. The module contains arrays of input and output gates, which sizes are specified at each instantiation of the bus. OMNeT imposes a restriction that not two modules can be connected to the same given gate. Therefore, if 3 computers plus a router are to be connected to the same Ethernet link, then the input and output gates are arrays of size 4.

3.3.1.3 The Sinks

A sink is a module that consumes packets. The sink consumes PDUs and keeps statistics about the number of frames received, the latency of each one, and number of collisions detected at the corresponding node. There is one sink per computer. The sink contains an input gate only.

```
simple Sink
gates:
  in: in;    // input gate
endsimple
```

Figure 6: File Sink.ned

3.3.1.4 The Routers

Each airplane encompasses three computer nodes and one router. The router is connected to the LAN, WPP and WSP links, as indicated in Figure 7. The LAN connection is direct because the bus module and the router module are connected without requiring any intermediate object. However, the connections to the wireless channels are indirect because the router is contained in the airplane, which is the intermediate object between the router and the wireless buses. Therefore, the `ned` specifications indicate that the router is directly connected to the airplane, which in turn is directly connected to the wireless links. Each connection requires one input and one output gate. The connections are:

1. Direct connection to the local Ethernet bus (100 Mbps)

2. Indirect connection to the wireless plane-to-plane bus (64 Kbps or more)

3. Indirect connection to the wireless plane-to-satellite bus (64 Kbps or more)

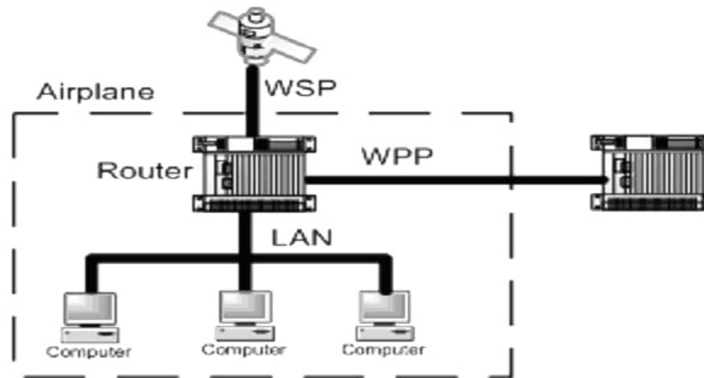


Figure 7: Router Onboard a Plane and its Connections to the LAN, WPP and WSP Links

Because all DIS traffic is broadcasted, a PDU originating from one of the input connectors is propagated to the other outputs according to Table 2. The `ned` source code defining a router is listed in Figure 8.

Table 2: Routing Table in Broadcast Mode

Input Link	**Output Link**
LAN	WPP and WSP
WPP	LAN
WSP	LAN

Any device connected to a bus gate must indicate its position measured in meters from one end of the bus. This position is a parameter involved in propagation delay calculations. The router is directly connected to the local Ethernet bus. For this reason, a `LANposition` parameter is required to establish the position of the router within the bus. The other bus connections are indirect because the router is really connected to a gate in the plane that, in turn, is connected to the bus. Therefore, positions for the wireless buses are indicated as parameters of the airplane, the compound module holding the LAN.

```
simple Router
parameters:
  routerID : numeric,
  nodesPerPlane: numeric,
  totalPlanes: numeric,
  LANposition : numeric, // Local LAN position
  routerServiceTime: numeric;
gates:
  in: inFromLocal;      // gate #0
  out: outToLocal;      // gate #1
  in: inFromWirelessPP; // gate #2
  out: outToWirelessPP; // gate #3
  in: inFromWirelessSP; // gate #4
  out: outToWirelessSP; // gate #5
endsimple
```

Figure 8: File Router.ned

Routers maintain an M/M/1 queue of input messages. Every time a new message arrives, the router records statistics about the number of messages in the queue at that time. The message length, the IFS gap, and the output bandwidth determine the service time, as indicated by the following formula:

$$T_s = L/B + \epsilon \quad (3.6)$$

where T_s, L, B, and ϵ denote the router service time in seconds, the message length in bits, the output bandwidth in *bps* and the *IFS* gap in seconds, respectively.

3.3.1.5 The Satellite

The satellite behaves like a router with only two links attached: the WSP and the WGS links. The distances from the satellite to the airplanes and from the satellite to the ground station were estimated in 38,300 kilometers: 35,800 Km of vertical distance, plus 2,500 Km of horizontal distance. The satellite also maintains a queue

of messages and calculates statistics as any other router does. Its `ned` source code can be seen in Figure 9. The parameter descriptions are similar to the parameters of a router, and so they are omitted here.

```
simple Satellite
parameters:
  satelliteID : numeric,
  satServiceTime : numeric,
  totalNodes : numeric,
  WGSposition : numeric, // Position at wirelessGS
  WSPposition : numeric; // Position at wirelessSP
gates:
  in: inBus1;   // gate #0 (wirelessGS)
  out: outBus1; // gate #1 (wirelessGS)
  in: inBus2;   // gate #2 (wirelessSP)
  out: outBus2; // gate #3 (wirelessSP)
endsimple
```

Figure 9: File Satellite.ned

3.3.2 Compound Module Definition

Compound modules are modules that contain other modules inside. For example, a computer onboard an airplane is a compound module because it contains a message generator and a sink. A plane is also a compound module that contains a computer, a router and an Ethernet bus. The largest compound module corresponds to the whole network that contains the airplanes, the satellite, the ground station, and the wireless buses linking these elements. Each one of these compound modules is briefly discussed in the following paragraphs.

3.3.3 Flying LAN Computer Nodes

Each workstation in the model consists of a computer node that contains two other submodules: the generator and the sink of PDUs. These computer nodes are directly connected to the LAN link. Figure 10 shows the OMNeT++ representation of a computer node, and Figure 11 lists its `ned` source code. The module `Node` is composed of the simple modules `gen` (generator) and `sink`. The module also contains an `in` (input) and an `out` (output) gate.

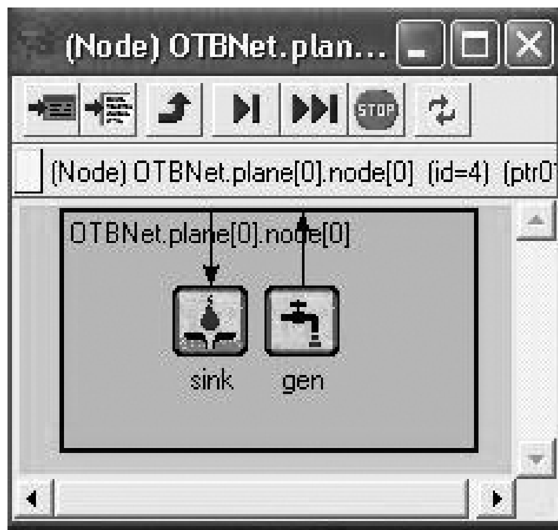


Figure 10: OMNeT Representation of a Computer Node and its Components

3.3.3.1 The Planes

The module `plane` is composed of the modules `router`, an array of `nodes` and the `ethernetBus`, as seen in Figure 12. The array length is one of the input parameters, set to 3 in this simulation. The corresponding `ned` file of this module is longer than the file of previous modules and is provided in Appendix B.

```
module Node
parameters:
  nodeID : numeric,
  LANposition : numeric;
gates:
  out: out;
  in: in;
submodules:
  gen: Generator;
parameters:
  fromAddr = nodeID,
  totalNodes = ancestor totalNodes;
  display: "i=gen;p=120,49;b=32,30";
  sink: Sink;
  display: "i=sink;p=81,49;b=32,30";
  connections:
  gen.out --> out;
  sink.in <-- in;
  display: "p=18,2;b=176,102";
endmodule
```

Figure 11: Ned Code of a Computer Node

The arrows in Figure 12 represent connections between modules via input and output gates. The router is also connected to the airplane input and output gates (not shown) which in turn are connected to two wireless buses.

3.3.3.2 Remote Ground Station

The ground station behaves exactly as any of the flying workstations. It is connected to the WGS link only. Figure 13 represents the ground station and Figure 14 shows its `ned` source code. This module is quite similar to the computer nodes and therefore a description is omitted.

Although the CONUS ground station technically is like any other flying workstation, it plays an important role in the simulator because it is the only station

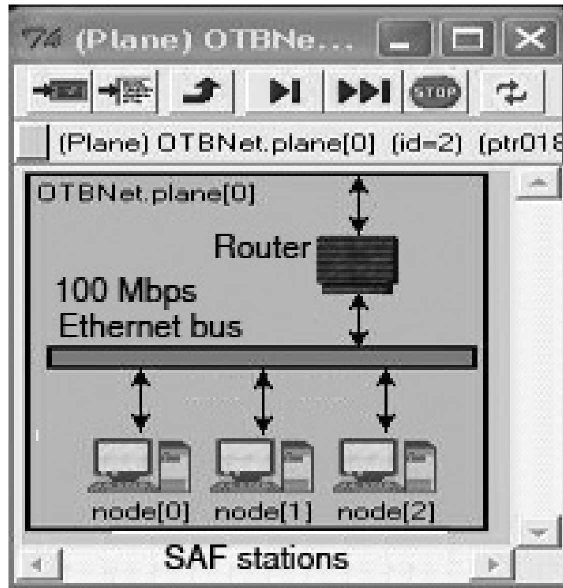


Figure 12: Airplane View Showing 3 Computer Nodes, a Bus and a Router

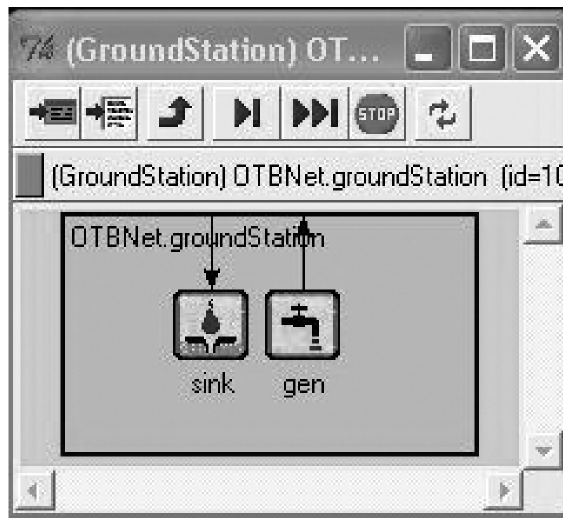


Figure 13: OMNeT View of the Ground Station and its Components

directly connected to a slow wireless link. The rest of the stations are connected to a 100 Mbps LAN link. Due to this characteristic, some of the simulations assigned the highest load in terms of number of generated PDUs to the ground station, and many statistics were collected around it.

```
module GroundStation
parameters:
  nodeID : numeric,
  WGSposition : numeric;
gates:
  out: out;
  in: in;
submodules:
  gen: Generator;
parameters:
  fromAddr = nodeID,
  totalNodes = ancestor nodesPerPlane * ancestor numPlanes;
  display: "i=gen;p=120,49;b=32,30";
  sink: Sink;
  display: "i=sink;p=81,49;b=32,30";
  connections:
  gen.out --> out;
  sink.in <-- in;
  display: "p=18,2;b=176,102";
endmodule
```

Figure 14: Ned Code of Ground Station

The parameter `fromAddr` of the generator comes from the parameter `nodeID` of the ground station. `TotalNodes` is defined as the product of `nodesPerPlane` and `numPlanes` that are defined at a higher level in the hierarchy of modules. The display feature indicates the position of this module in the graphical user interface (GUI). Finally, the module establishes the connections between the gates from the inner (simple) modules and the container module (the ground station).

3.3.4 Instantiation of the Network

The compound module `TheNet` contains the submodules `wirelessPP`, `wirelessSP`, `wirelessGS`, `groundStation`, `satellite`, and `plane` as shown in Figure 3.3. Due to its length, the source code of the `ned` language for this module is found in Appendix B.

OMNeT++ uses the object programming approach. The modules as well as the whole network are considered classes that must be instantiated. In this simulation `TheNet` is the class name of the network that is instantiated as `Network OTBNet`. It includes all of the model parameters that are read from the `omnetpp.ini` file. Figure 15 shows this instantiation and the corresponding parameters.

```
network OTBNet : TheNet
parameters:
  startTime = input,    //First PDU timestamp in seconds
  nodesPerPlane = input,//Set to 3 in this simulation
  numPlanes = input,    //Set to 8 in this simulation
  LANgapTime = input,   //Minimum gap between frames in LAN
  LANbandwidth = input, //Set to 100 Mbps
  LANdelay = input,     //nanosec/meter (70% light speed)
  WPPgapTime = input,   //Minimum gap between frames in WPP
  WPPbandwidth = input, //Wireless bandwidth in WPP
  WPPdelay = input,     //nanosec/meter (light speed)
  WSPgapTime = input,   //Minimum gap between frames in WSP
  WSPbandwidth = input, //Wireless bandwidth in WSP
  WSPdelay = input,     //nanosec/meter (light speed)
  WGSgapTime = input,   //Minimum gap between frames in WGS
  WGSbandwidth = input, //Wireless bandwidth WGS
  WGSdelay = input,     //nanosec/meter (light speed)
  satServiceTime = input, //Satellite service time
  routerServiceTime = input; //Router service time
endnetwork
```

Figure 15: Instantiation of the Network `TheNet`

The initialization file `omnetpp.ini` is shown in Figure 16 for an OMNeT model. It is used to specify input parameters to the model. The parameter `output-vector-`

`file` is used to specify the output file that contains all the simulation results. At the end of the simulation, this ASCII file is processed by any application capable of interpreting it and producing statistics and/or graphics. OMNeT provides the `plove` plotting tool to process this kind of vector file. The parameter `sim-time-limit` is used to indicate an upper limit to the simulation time. Similarly, `cpu-time-limit` gives an upper limit to the CPU time used by the simulator.

The parameter `display-update` is used for the simulation with animation, and indicates the refreshing rate of the window. The section under `[Run 1]` contains all the parameters specific to a given run. It is possible to indicate several run sets with different parameters each by appending sections `[Run 2]`, `[Run 3]`, etc. A more detailed explanation of the initialization file is found in the OMNeT User Manual [Var03].

3.4 Messages in the Simulator

The PDUs that conform the network traffic are logical objects embedded in physical messages of the simulator. The physical messages consist of a data structure defined by OMNeT. Some of the attributes in a physical message are *name*, *length*, and *timestamp*. The name is used to identify the message and is considered its type. The length is set to the PDU length in bits, and the timestamp is used to record the simulation time when the message is released to the network. The message types used in the simulator are the following:

***BlockTimeout*:** When a generator receives this message, the oldest PDU in the aggregation block has timed out and the block must be send as soon as possible.

```
[General]
network = OTBNet
ini-warnings = no
random-seed = 1
warnings = yes
snapshot-file = planes.sna
output-vector-file = planes64repl.vec
sim-time-limit = 2550s  # simulated seconds (42:30)
cpu-time-limit = 20h    # 20 hours of real cpu time max.
total-stack-kb = 4096   # 4 MByte, increase if necessary
[Cmdenv]
module-messages = yes
verbose-simulation = yes
display-update = 0.5s
[Tkenv]
default-run=
use-mainwindow = yes
print-banners = yes
slowexec-delay = 300ms
update-freq-fast = 10
update-freq-express = 100
breakpoints-enabled = yes
[DisplayStrings]
[Parameters]
[Run 1]
OTBNet.startTime     = 1034s   # 17:14
OTBNet.nodesPerPlane = 3
OTBNet.numPlanes     = 8
OTBNet.LANgapTime   = 50us
OTBNet.LANbandwidth = 100E6    # 100 MBps
OTBNet.LANdelay     = 4.761904762ns #nsec/meter, 70% light sp
OTBNet.WPPgapTime   = 50us
OTBNet.WPPbandwidth = 64000
OTBNet.WPPdelay     = 3.333333333ns #nsec/meter, light speed
OTBNet.WSPgapTime   = 50us
OTBNet.WSPbandwidth = 64000
OTBNet.WSPdelay     = 3.333333333ns #nsec/meter, light speed
OTBNet.WGSgapTime   = 50us
OTBNet.WGSbandwidth = 64000
OTBNet.WGSdelay     = 3.333333333ns #nsec/meter, light speed
OTBNet.satServiceTime = 5us
OTBNet.routerServiceTime = 5us
OTBNet.generatorServiceTime = 5us   #PDU bundling time
OTBNet.blockWaitTime = 100ms
```

Figure 16: Initialization File Omnetpp.ini

***ReadyToSend*:** This message indicates that the generator has finished transmitting a previous PDU, and at this time it is ready to accept new PDUs.

***Data*:** This message contains the PDU or block of PDUs. The message name is actually a C string of characters with the format "Data%d Id%d P%d F%d T%d". The fields in the message are a frame counter, the PDU ID, PDU priority, originating site number, destination site number. The destination number is -1 in all cases indicating that the message will be broadcasted. For example, "Data10 Id25 P5 F0 T-1" indicates that this PDU is the tenth frame sent by this site containing PDU number 25 as indicated in the summary file, the priority is 5, site 0 is the sender and all the other sites are destinations. The information in the message name is visible on the computer screen when the simulator is executed at low speed with the animation feature on.

This Chapter showed the design and implementation of all the components in the communication architecture for the OMNeT simulator. The actual C++ code in each module depends on the strategies used for bundling and queuing the PDUs. These issues will be discussed in the following Chapters.

CHAPTER 4

PACKET ALLOYING BUNDLING TECHNIQUE

Compression and aggregation of network packets are techniques used to reduce the total traffic, requiring less bandwidth for sending data. During the bundling process, decisions have to be made about whether to bundle or not bundle consecutive packets. Considering that after one packet has been sent, a certain minimum time gap must elapse before sending the next packet, then the decision is not trivial. If the first packet is sent immediately, the second one could be delayed more than if the two packets are sent in a single bundle. This decisions are referred to as the *Packet Bundling Problem* [FL02]. In the case of an OTB simulation, the offline bundling is carried out taking as input the PDU log files produced by the OTB simulator, and not only the time gaps, but also the type and the length of the PDUs are considered.

4.1 Characteristics of Embedded Simulation Traffic Impacting Bundling

Several characteristics of the Embedded Simulation traffic are to be considered for bundling purposes. The discussion here applies to PDUs generated under the

DIS protocol, although many of them are general enough to be valid under other protocols as well.

4.1.1 Real-Time Dynamics

During an Embedded Simulation, the participants interact with each other in real time. If one vehicle starts moving, or decelerates, all the other entities should be informed of the event as soon as possible. This characteristic impacts bundling in several aspects. First, the time a PDU waits for the upcoming PDU creates a delay against the meaning of real time. Therefore, a small timeout should be introduce to limit that waiting time. Second, not bundling PDUs could cause that the following PDU will be delayed even more, due to the gap time that must separate frames. Also, not bundling produces more traffic and longer router queues, which finally goes in detriment of the real time properties.

The decision of bundling PDUs must consider the pros and cons of each alternative. It is possible that for some environments bundling is not a necessity, for example a scenario in which few sites are simulating a simple vignette, connected in a LAN, not requiring a router.

4.1.2 High Ratio of ESPDU Traffic

It is well documented that Embedded Simulation traffic contains 70 % or more of Entity State PDUs (ESPDUs) [MZP94, Mac95, SZB96, BCL97]. Usually these ESPDUs are partially redundant and not urgent. For example, if a vehicle is not moving, it still needs to send heartbeat ESPDUs at regular intervals. ESPDUs are

less impacted by waiting to be bundled than other PDUs of higher priority. Because they are so abundant, bundling and compressing ESPDUs have a major impact on the overall traffic decrease. However, in this research ESPDUs were not bundled in the majority of cases because they did not participated in negative spikes as bursts of consecutive ones, as `po_fire_parameters` did.

4.1.3 Low Volume of High Priority PDUs

Some high priority PDUs like fire and detonation occur in short bursts, but they are usually sent at the same time, creating negative slack spikes that attempt against the real time approach. If the bundling operation is not time-consuming and the waiting timeout is selected appropriately, bundling a sequence of consecutive PDU and sending a single block could take less time than sending the individual PDUs without bundling them. For instance, if a PDU is 512 bytes long and the bandwidth is 64 Kbps, the transmission time of one single PDU is 64 milliseconds, while bundling several PDUs could take less than one millisecond.

4.1.4 Transmission Redundancy

PDUs may contain redundant field data, both inside each PDU and among PDUs. As a sample, the PDUs listed in Table 3 contain zeroes in the majority of fields, and just two differences from one PDU to the other. Bundling and compression can take advantage of this high redundancy. For example, the proposed algorithm Packet Alloying would append only the two fields containing the differ-

ences in the second PDU of the table to the first one. Other bundling algorithms also profit from this redundancy, like those based on sending delta PDUs.

4.1.5 Regular Packet Structure

PDUs have a definitive structure made up of fields of different sizes, that are determined by the type and length of the PDUs. This characteristic allows the comparison of PDUs at the field level, which facilitates the extraction of the differences. Determination of the PDU structure based on the type and the length allows a fast comparison among PDUs for algorithms like Packet Alloying.

4.1.6 Broadcast Transmission

In the DIS protocol, PDUs are broadcasted. This characteristic simplifies the PDU header since a particular destination is not needed. All the PDUs in a bundle are to be delivered to the same recipients. This simplifies the bundling and routing process accordingly. However, broadcasting contributes to the proliferation of messages sent to entities that might not need updated information from all the other entities, some of which could be very far away in the simulation field, and the non reception of such messages is not going to significantly affect the simulation fidelity. Instead of broadcasting, multicasting is an alternative proposed by other protocols like HLA and DIS-Lite [VCR96, SH96, CTH02].

4.1.7 Low Bandwidth

The slower the connections, the more significant the impact of bundling is. If a connection is slow, bundling and compression becomes more advantageous. In slow connections, gap times are larger, and so is the penalty for not bundling the next PDU. Also, a low bandwidth connection shows more negative slack times during transmission, causing bottlenecks and long queues. As an example, in the MR1 vignette the satellite connection introduces a propagation delay of about 0.25 seconds, much higher than the time gap required to separate frames during transmission. Therefore, bundling of high priority PDUs like `po_fire_parameters` tends to be worthwhile as shown in Chapter 5.

4.1.8 Simultaneous Scheduling of PDU Bursts

Bursts of PDUs timestamped at the same or almost the same time encourages benefits of bundling, because not doing it causes a large negative slack spike, or bottleneck, at the transmitting site that delays the following PDUs. Essentially, instantaneous transmissions overwhelm the channel capacity so that the available bandwidth appears low relative to the demand.

4.2 Offline and Online Algorithms

General offline and online algorithms for predictive environments have been studied in the literature for a considerable time [Kar92, FL02, FLN03, GHP03]. Bundling strategies are particular cases of the general algorithms that are confronted with the

decision of waiting for the next packet to arrive and bundle it to the current block, or sending the current block and start a new collection of packets from scratch. This type of decisions are typical of *online* and *offline* aggregation algorithms.

Definition 4.1: ***Offline Algorithm***.

> Let σ be a sequence of input data, in which the decision of processing each individual data element $\sigma_i \in \sigma$ results in applying one of several possible actions. If the selected action is taken using the complete knowledge of the whole sequence σ, the selection process constitutes an *offline algorithm*.

Because offline algorithms know the complete packet sequence at the decision time, including the future packet sequence, the best offline algorithm can make the optimal decision. A decision is optimal if it minimizes some *cost function*. In the transmission of network packets, the cost function could be the total latency time incurred by all the packets sent from the origin to the final destination. Another cost function could be the absolute value of the sum of all negative slack times when the PDUs are sent.

Definition 4.2: ***Total Latency Cost***.

> Symbolically, given a sequence $\sigma = \{PDU_i\}_{i=1,\ldots,n}$ of PDUs, where each packet i is released and stamped at time $Tstamp_i$, and arrives at the final destination at time $Tarr_i$, then the cost function for the total latency of the travel time is:

$$C_{Ttrav}(\sigma) = \sum_{i=1}^{n}(Tarr_i - Tstamp_i) \tag{4.1}$$

Similarly, if the simulator is reading PDUs from a summary log file, and each PDU_i is read for the first time at time $Tread_i$, the cost function that measures the

absolute value of the total negative slack time is:

$$C_{Tslack}(\sigma) = \sum_{i=1}^{n} (Tstamp_i - Tread_i) \times H(Tread_i - Tstamp_i) \tag{4.2}$$

where H represents the Heaviside step function:

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{4.3}$$

used to select only the negative slack occurrences.

Offline algorithms are useful in many other computer-related areas. Most of the database algorithms like sorting and searching files are offline. A recent example of an offline algorithm for compressing data is given by Turpin in [TS02]. Offline algorithms are not required to be implemented in an actual real-time simulation because obviously the messages to be produced in future times are not known at the present simulation time. In simulation, the main application of offline algorithms lies in the possibility of comparing them to the corresponding online counterparts, with the purpose of assessing online performances. A measure of comparison of performance for online algorithms is the *competitive ratio*. A competitive ratio of $r > 0$ means that the performance of an online algorithm is at least a factor of $1/r$ of the performance achieved by the best offline algorithm. This will be expanded in the next section.

Online algorithms are characterized by a lack of knowledge about the future. Phillips [PW99] indicates that the online algorithm receives each input in sequence and must process it immediately, serving the sequence of requests one item at a time without having explicit knowledge of the following inputs. Karp defines these algorithms in the following way:

Definition 4.3: ***Online Algorithm***.

An online algorithm is one that receives a sequence of requests σ and performs an immediate action in response to each request σ_i before receiving σ_{i+1}. Online algorithms arise in any situation where decisions must be made and resources allocated without knowledge of the future [Kar92].

The effectiveness of an online algorithm may be measured by its *competitive ratio*, defined as the worst-case ratio between its cost and that of a hypothetical offline algorithm which knows the entire sequence of requests in advance and chooses its actions optimally (with minimum cost). The performance of online deterministic algorithms is measured by its competitive ratio when compared with the optimal offline algorithm. Ambühl [AGS01] defines this ratio in the following way:

Definition 4.4: ***c-Competitiveness***.

If σ is any input sequence, $A(\sigma)$ represents the cost function of an online algorithm A, and $OPT(\sigma)$ is the corresponding cost function of the optimal offline algorithm OPT, then A is called *c-competitive* for a constant c if there exists a real number a such that for all input sequences σ, it is true that:

$$A(\sigma) \leq c \cdot OPT(\sigma) + a \tag{4.4}$$

Definition 4.5: ***Strictly c-Competitiveness***.

If $a = 0$ in equation 4.4, then A is called *strictly c-competitive.*

If A is a randomized algorithm, equation 4.4 becomes:

$$E[A(\sigma)] \leq c \cdot OPT(\sigma) + a \tag{4.5}$$

where $E[A(\sigma)]$ represents the expected cost of algorithm A on the sequence σ.

The competitive ratio of an algorithm is defined as the infimum over all real numbers c such that the algorithm is c-competitive [FLN03, GIS03]. The competitive ratio has been discussed recently and values for some algorithms have been calculated. Phillips [PW99] proves that the well known Least-Recently-Used (LRU) and First-In-First-Out (FIFO) algorithms for paging in virtual memory systems have competitive ratio k, where k is the number of pages that can be stored in main memory. He also says that on traces taken from program executions, the performance ratio of LRU is much less than k, typically close to 2. Frederiksen [FL02] indicates that any reasonable deterministic or uniform randomized algorithm for packet transmission has a competitive ratio of exactly 2, where an algorithm is called reasonable if it does not postpone the transmission of a message by more than the sum of the inter-packet gap and overhead values. Deorowicz [Deo03] point out that there are bounds for the competitiveness of the deterministic as well as randomized methods, and the lower bound for deterministic methods is $c = 2 - 2/(L + 1)$, where L is the length of the sequence σ. He also indicates that randomized methods can improve this bound.

4.3 Packet Alloying Modes

If several PDUs are scheduled at the same or almost the same time, and the structure of those PDUs is the same, with only small but predictable differences, then only one single PDU needs to be sent together with instructions on how to recover the other PDUs from the given one. Comparisons of `po_fire_parameters` type of PDUs among other PDUs involved in the same negative spike showed that the stated conditions (same timestamp, small differences) can be exploited. These

PDUs differ on consecutive identification attributes (like counters), and memory addresses that change according to the PDU length.

Table 3 shows two consecutive `po_fire_parameters` PDUs, identified as PDUs #19855 and #19856 in the OMNeT simulator and captured at second 1577.697 of simulation time, which are contributors to the negative spike at second 1577 seen at the righthand side in Figure 57 of Section 5.6.6. The table is quite long, and only highlights are shown here. The bundling method called Packet Alloying described in Section 4.3.1 was proposed after analyzing this comparison. The table shows that the two PDUs are almost identical, and most of their fields are zeros. Besides the address associated with each field, only two differences were found, highlighted in grey for the second PDU. The shown PDUs are not an exceptional coincidence. In all of the negative spikes studied, the participating PDUs have similar redundancies, provided that they are of the same type and length.

The observations and analysis of those PDUs participating in negative spikes lead to the proposal of Packet Alloying, a new scheme for bundling PDUs that can be seen as a kind of high-level lossless compression because the resulting block still conserves the characteristics of a PDU, perhaps of a different type, and so it is subjectable to further compressions. In fact, the proposed bundling does not remove the redundancy within the same PDU: the fields filled with zeros in the reference PDU will continue being the same length of zeros. Only the redundancy resulting from the similarities between consecutive PDUs is removed. Therefore, other traditional compression mechanisms are applicable and recommended after alloying.

Extraction is the inverse procedure of alloying. Given a bundled block that arrived to a destination, the individual PDUs have to be *extracted* or *replicated* from it. Extraction is independent of other data compression techniques because it is targeted at the PDU-level and the resulting traffic is of PDU type. Therefore,

Table 3: Comparison of Two Consecutive Po_fire_parameters PDUs

PDU Field	PDU 19855	PDU 19856
<dis204 po_fire_parameters PDU>:		
dis_header.version	0x04	0x04
dis_header.exercise	0x01	0x01
dis_header.kind	0xec	0xec
dis_header.family	0x8c	0x8c
dis_header.timestamp	26:17.697	26:17.697
dis_header.sizeof	0x0220	0x0220
po_header.po_version	0x1c	0x1c
po_header.po_kind	0x02	0x02
po_header.exercise _id	0x01	0x01
po_header.database _id	0x01	0x01
po_header.length	0x0210	0x0210
po_header.pdu_count	0x23ff	0x2400
do_header.database _sequence_number	0x00000000	0x00000000
do_header.object _id.simulator.site	0x05fc	0x05fc
do_header.object _id.simulator.host	0xb95b	0xb95b
do_header.object _id.object	0x0b52	0x0b52
do_header.world _state_id. simulator.site	0x0000	0x0000
do_header.world _state_id. simulator.host	0x0000	0x0000
do_header.world _state_id.object	0x0000	0x0000
do_header.owner.site	0x05fc	0x05fc
do_header.owner.host	0xb95b	0xb95b
do_header.sequence _number	0x0011	0x0012
do_header.class	0x13	0x13
do_header.missing _from_world _state	0x00000000	0x00000000
reserved9	0x00000000	0x00000000
fire_parameters.unit.simulator.site	0x05ef	0x05ef
fire_parameters.unit.simulator.host	0xb89f	0xb89f
fire_parameters.unit.object	0x0aa6	0x0aa6
fire_parameters.fire_zone[0].simulator.site	0x0000	0x0000
fire_parameters.fire_zone[0].simulator.host	0x0000	0x0000
fire_parameters.fire_zone[0].object	0x0000	0x0000
fire_parameters.fire_zone[1].simulator.site	0x0000	0x0000
fire_parameters.fire_zone[1].simulator.host	0x0000	0x0000
fire_parameters.fire_zone[1].object	0x0000	0x0000
...	...	...
fire_parameters.vehicle_def[14].engagement_range	0x00000000	0x00000000
fire_parameters.vehicle_def[14].ignore_after	0x00000000	0x00000000
fire_parameters.vehicle_def[14].priority	0x0000...0000	0x0000...0000
ffire_parameters.vehicle_def[15].engagement_range	0x00000000	0x00000000
fire_parameters.vehicle_def[15].ignore_after	0x00000000	0x00000000
fire_parameters.vehicle_def[15].priority	0x0000...0000	0x0000...0000

even if there are no plans to modify the transport protocol in effect (by compressing TCP/IP headers, for instance), the reduction of PDU packets to increment the bandwidth availability by using replication is still applicable.

After running the simulator without using bundling, and collecting performance statistics, it was observed that at 64 Kbps, the generator in ground station could not cope with the demanding traffic, and an increasing delay in timeliness to send PDUs at the indicated timestamp started to accumulate. The proposed solution was to bundle PDUs of the same type and length into longer ones, eliminating redundancy in similar fields, as explained in the following section.

4.3.1 Mathematical Description of Alloying

The following definition is named *Packet Alloying* in this dissertation, and is the basis for the bundling algorithm proposed in Section 4.4.

Definition 4.6: ***Packet Alloying***.

> Let $N = \{1, 2, \ldots, n\}$ be a set of indices, and let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ be two consecutive PDUs, where A and B are of the same type and the a_i and b_i represent PDU fields. For a subset $S \subseteq N$ such that $a_i = b_i$ for all $i \in S$, the bundle of A and B is defined as the PDU $A \otimes B = (a_1, a_2, \ldots, a_n,\ [(b_j, j)_{j \in N \setminus S}]\ )$. A is called the *reference* PDU in the bundle.

In the above definition, the notation $N \setminus S$ represents the set difference of N and S, defined by $N \setminus S = \{x : x \in N \text{ and } x \notin S\}$. The square brackets [,] denote the start and end, respectively, of replicated to be formed during extraction. The definition can be extended to any number of PDUs.

Example 4.1

> Given the PDUs: $A = (a_1, a_2, a_3, a_4)$, $B = (b_1, b_2, b_3, b_4)$, and $C = (c_1, c_2, c_3, c_4)$, such that $a_2 = b_2 = c_2$, $a_3 = b_3$, $a_4 = c_4$, then the *Alloying* $A \otimes B \otimes C$ is the new PDU
> $A \otimes B \otimes C = (a_1, a_2, a_3, a_4,\ [(b_1, 1), (b_4, 4)],\ [(c_1, 1), (c_3, 3)]\ )$

From the information contained in the shown n-tuple it is possible to reconstruct the original PDUs A, B, and C. Each component (b_j, j) indicates that the value b_j replaces the field j in the reference PDU. In a practical implementation, j could be a pointer or an offset into the reference PDU.

The above bundle will be called a *Packet Alloying* because it is formed like a metal alloy, bundling PDUs based on their internal structure. It differs from other proposals in several ways. First, the resulting bundle conserves the basic characteristics of any other PDU and therefore, can be considered a sub-type of PDU subjectable to further bundling and/or compression algorithms. In [BCL97] consecutive PDUs are concatenated in a single packet even if their types are different, and field redundancy is not eliminated. A delta-PDU encoding technique is mentioned in [US95a] consisting of PDUs that carry changes respect to a reference PDU initially given. In [WMS01] several bundling techniques generally applicable to Web pages under the TCP/IP protocol suite are described, but none is specific to the DIS protocol. A protocol called DIS-Lite developed by MäK Technologies [Tay95, Tay96b, Tay96a, PW98] splits the Entity State PDU into static and dynamic data PDUs, so that the static information is sent once and the changes (dynamic PDUs) are subsequently sent as separate PDUs. DIS-Lite includes also several other improvements not related to the combination of individual fields from a set of similar PDUs.

4.3.2 Online Bundling Strategies

The online algorithms proposed try to identify compatible PDUs P_1 and P_2 that can be bundled in a block $B = P_1 \otimes P_2$. A requirement is that the two PDUs should have the same type and length. Yet, because P_2 has not arrived by the time P_1 is being processed, the generating site must decide whether it will send P_1 immediately, or wait for P_2. Chances are that P_2 will not be compatible with P_1. So if the generator could predict the type and the length of P_2, then the prediction could be used in the decision process. It is more difficult to predict both, type and length, than only one variable. For decisions based on one variable only, `type` is preferred because it is shown to discriminate better among PDUs, as will be shown below.

4.3.2.1 ***Always-Wait*** and ***Always-Send*** Type Predictions

Two straightforward online algorithms were proposed to predict the type of the next PDU: *Always-Wait* and *Always-Send*. The former predicts that the next type will be the same as the type of the current PDU and takes the decision of wait in each case, using a timeout of 100 milliseconds. This value is much less than the 0.25 second delay of the satellite link. The latter predicts a type different from the current PDU type and never waits. It sends the PDU as soon as the time gap has elapsed without bundling PDUs. A third online prediction method based on a neural network approach used to predict the PDU type is explained in the next section.

4.3.2.2 PDU Type prediction using a Neural Network

One means of predicting the next PDU based on the recent history is by using a Neural Network (NN) approach. A NN can learn sequences of PDU patterns and use them as a basis for predicting the incoming type. In this research, we set up a *gradient descent* NN that predicts the next type based on the types of the previous 44 PDUs. This number was chosen because the longest sequence of consecutive PDUs of the same type was found to be 42, and we wanted to have at least 2 different PDU in the sequence. The NN architecture contains 44 input nodes, 20 hidden nodes, and 5 output nodes that specify the predicted type using a binary representation. The population of PDUs was split in two equal size sets of 25,093 PDUs each for training and prediction. Once the sequences of 44 PDU types plus the predicted type were prepared, they were randomly sorted for a better learning in the training phase. The program run for several days, and after 11,930 epochs the percentage of successful predictions reached almost 70 %. Considering that there are near 27 different PDU types, this percentage is meaningful and significant.

In order not to unnecessarily complicate the simulator logic, the NN procedure was run offline, and the results were incorporated into the PDU summary file. If the NN prediction indicates that the next PDU type is the same as the current one, a `W` character meaning *Wait* is appended offline to the summary file, otherwise an `S` character meaning *Send* is appended, as will be discussed later.

4.3.3 Offline Bundling Strategy

Due to the fact that the summary PDU files used with the simulator do not include the actual PDU fields, it is not possible to determine the bundling resulting

from two summary PDUs of the same type and length because their fields cannot be compared and their differences cannot be established. In this research, the comparison took place offline in the pre-processing stage using actual PDU fields, which resulted in three *ideal prediction* offline methods, additionally to the online methods. The ideal prediction methods calculate the next PDU type with 100 % certainty because they know the entire sequence of PDUs in advance.

4.3.3.1 Predictions Based on Type, Length, and Timestamp

The first ideal prediction method considers the PDU type only and is called *Type* prediction. If two consecutive PDUs are of the same type, then this method predicts *Waiting* and a `W` character is appended to the description of the current PDU in the summary file, otherwise an `S` for *Sending* is appended. The second method considers the type and length of the PDUs and is called *Type-Length*. The `W` character is appended to the summary file if both type and length are equal in two consecutive PDUs, and and `S` is appended otherwise. The last method requires the same type, length, and timestamp of two consecutive PDUs and is called *Type-Length-Time*. More formally, considering the functions:

- $type(PDU)$
- $length(PDU)$
- $timestamp(PDU)$
- $compatible(PDU_1, PDU_2)$

that return the type, length and timestamp of any PDU, as well as a Boolean *TRUE* or *FALSE* value of PDU compatibility, respectively, then three offline bundling algorithms can be defined: *Type*, *Type-Length*, and *Type-Length-Time*, as explained next.

Definition 4.7: ***Type Delimited Alloying***.

If $(P_i)_{i=1...n}$ is the sequence of PDUs to be transmitted from some site, then the *Type* offline algorithm makes its bundling decision based on the type of the PDUs. Using the notation of Section 4.3.1, if the block B already contains some PDUs, being P_k the first one (reference PDU), then the decision of bundling P_j to B ($B = B \otimes P_j$) will be taken if $type(P_j) = type(P_k)$.

If the two PDUs bear the same type, then during the simulation their compatibility is analyzed. If $compatible(P_j, P_k)$ is $TRUE$ and the timeout period has not expired, then the PDUs are bundled.

Definition 4.8: ***Type-Length Delimited Alloying***.

The *Type-Length* offline algorithm makes its bundling decision based on both the type and length of the PDUs. Given the sequence $(P_i)_{i=1...n}$ of PDUs, and assuming that the block B contains the reference PDU P_k, then the decision of bundling P_j to B ($B = B \otimes P_j$) will be taken if $type(P_j) = type(P_k)$ and $length(P_j) = length(P_k)$.

If the two PDUs bear the same type and length, then during the simulation time their compatibility will be analyzed, producing a successful comparison. If the timeout period has not expired, then the PDUs are bundled.

Definition 4.9: ***Type-Length-Time Delimited Alloying***.

This last scheme called *Type-Length-Time* alloying is similar to the previous methods, except that it includes an extra condition for bundling: $timestamp(P_j) = timestamp(P_k)$. This condition may first appear to be very restrictive because it asks for an exact match between timestamps in both PDUs. If two PDUs were scheduled with a time difference of one single nanosecond, this offline algorithm will not bundle

them. In practice, it has been observed that OTB can schedule hundreds of PDUs at exactly the same time. Therefore, the purpose of this algorithm is to evaluate the impact of bundling just those PDUs.

Figure 20 in Chapter 5 shows four characters at the end of each line containing the `S` and `W` designations. The first character corresponds to the neural network prediction scheme and the remainder correspond to each one of the ideal prediction methods. Characters corresponding to *Always-Wait* and *Always-Send* are not stored, but assumed by considering that one column of the summary file is filled all with either `W` or `S` characters, respectively. Only one character is processed per execution of the simulator and the `omnetpp.ini` file is set up to include the prediction method desired in each run.

Chapter 5.8 shows that the Packet Alloying techniques described here can have a significant impact on the communication performance of an embedded simulation by reducing the network traffic and satellite queue length up to 88 % in some cases.

4.4 Development of Alloying in Simulation Model

After analyzing all of the PDUs in the log file for a given vignette, it was observed that the type and the size of a PDU can adequately determine its internal field structure.

Definition 4.10: ***Compatible PDUs***.

Two PDUs $A = (a_1, a_2, a_3, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ are said to be *compatible* if and only if $type(A) = type(B)$ and $length(A) = length(B)$, assuming that *type* and *length* are functions that return the type and the length in bytes of a PDU, respectively.

Conjecture 4.1

> If the PDUs $A = (a_1, a_2, a_3, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ are compatible, then $n = m$ and $\mathit{field_type}(a_i) = \mathit{field_type}(b_i)$ for all $1 \leq i \leq n$, assuming that *field_type* is a function that returns the type of any field in the PDU.

Conjecture 4.1 cannot be proved unless the formal specifications of PO_PDUs are analyzed and the OTB source code is examined, items not made available to this research. However, as verified empirically by an analysis program, all of the 60,341 PDUs in the MR1 vignette were analyzed, as well as the PDUs in other two vignettes, and no exceptions to Conjecture 4.1 were found. Because compatible PDUs share a common internal structure, they are good candidates to be bundled. The other requirements to deliver the PDUs as a single packet are that they must be consecutive and scheduled within a short time interval, as defined below.

The pseudo-algorithm of PDU bundling is described in Figure 17. This algorithm corresponds to Always-Wait alloying because after processing a PDU, it waits for the next PDU unless a timeout is detected. When the next PDU is obtained, its type is checked and if it is different from the type of the reference PDU, the reference PDU is sent and the time waited is wasted. Yet, if an online algorithm is applied to predict the type and length, or at least the type of the next PDU and the prediction indicates a type different form the one in the bundle, then the current bundle could be sent immediately, saving the waiting time.

Correctness of Pseudo Algorithm in Figure 17

> The goal of this pseudo algorithm is to bundle consecutive PDUs if they are compatible and the oldest PDU in the bundle has not timed out. If the conditions are not met, i.e. the next PDU is not compatible or the oldest

```
1. Wait until (next PDU is ready for delivery)
       Let A denote that PDU;
2. Block = A;    /* This is the first PDU
                    (reference) in the bundling */
3. Set timeout =  maximum time A will wait in Block;
4. While (timeout not expired)
5.       {  If (next PDU is ready for delivery)
                Let B be that PDU;
            else Continue at the While-loop;
6.          If (A and B are compatible PDUs)
               { Block = Block (x) B; /* Packet Alloying bundling */
                 B = empty;
               }
            else break of the While-loop;
         }
7. Send Block through the network as a single packet;
8. If  (B is empty)
       Continue at Step 1;
   else { A = B;
         Continue at Step 2; }
```

Figure 17: Pseudo-Algorithm of PDU Bundling

PDU has timed out, then the already built bundle must be sent as soon as possible and the next PDU will start a new bundle.

Starting with an empty bundle called *Block*, steps 1, 2, and 3 in Figure 17 ensure that the bundle has been initialized and the timeout of the oldest PDU in the bundle, which corresponds to the first PDU in A, has been set.

Steps 4, 5, and 6 are executed while the timeout is not expired. Within this `While-loop` every new PDU B released for delivery is compared to the first PDU (A) in the bundle, and if the new PDU is compatible with A, it is bundled to *Block*, emptying the variable B. Because A is fixed during the while loop, all the PDUs bundled are compatible with A. Being compatible is an equivalence relation, and so all the PDUs in the bundle are compatible among each other. The `While-loop` is finished by one of two possible reasons: a new PDU not

compatible with the bundle is found, or the oldest PDU in the bundle has timed out. In either case, Step 7 is executed.

Step 7 sends the bundle and Step 8 analyzes the reason for breaking the loop. If B is empty, the `While-loop` was broken by a timeout and the algorithm has to wait for the next PDU to be available to start a new bundle in Step 1. Otherwise the previously read PDU was not compatible with the *Block*, but because the PDU is in A, Step 2 must be executed, bypassing the waiting time of Step 1.

Thus, the pseudo algorithm in Figure 17 complies with the specifications in the goals for this alloying strategy.

Performance of Pseudo Algorithm in Figure 17

The performance of this pseudo algorithm depends on the implementation, but some time complexity relationships can be derived. PDUs are read in either Step 1 or Step 5. Assuming a sequence of n PDUs in an entire vignette, the algorithm processes each PDU sequentially. This gives a time complexity of $O(n)$, if the time required for bundling and sending blocks is bounded by a constant value. If the bundling operation is not considered bounded, but dependent on a maximum number k of fields in the PDUs, then each bundle operation will take $O(k)$ time to compare the fields in the new PDU to those in the reference PDU A. In that case, the time complexity of the pseudo algorithm is $O(kn)$.

Another performance issue has to do with the convenience of bundling compatible PDUs in all situations. According to the definition, each field in the new PDU that differs from the corresponding field in the reference PDU is appended to the bundle along with an index. In the extreme case that all the fields in the new PDU are different from the corresponding fields in the reference PDU, the new PDU will be

bundled with no reduction in size due to a lack of redundancy elimination. Moreover, the inclusion of indices would make the bundle larger than the sum of the sizes in the two individual PDUs in this case. A threshold parameter for comparing the size of the bundled fields and indices to the size of the unbundled PDU to decide whether the bundle is efficient or not is a performance enhancement for this algorithm.

Figure 18 shows the general bundling algorithm used by the generators for sending PDUs to the network, including prediction information taken from the summary files. The abbreviations used in the figure are:

- `BTO`: Block timeout indicating that the oldest message in the block has timed out
- `BWT`: Block waiting time, delayed incurred due to bundling, set to 100 milliseconds in the simulation
- `EOF`: End of file
- `GT`: Gap time, minimum separation between packets, set to 50 microseconds in the simulation
- `MSG`: Message of any type that arrives to the generator
- `Now`: current simulation time
- `RTS`: Ready-to-send message indicating that the generator can accept new messages
- `ST`: Service Time of the generator, set to 5 microseconds in the simulation
- `TT`: Transmission time, dependent on the PDU length and bandwidth

The generator is activated when it receives one of two possible messages from the OMNeT kernel:

BlockTimeout (`BTO`): indicates that the oldest PDU in the current bundle has timed out (`BWT` has elapsed), and therefore the generator must send it as soon as possible. This means that if the generator is idle, then the bundle can

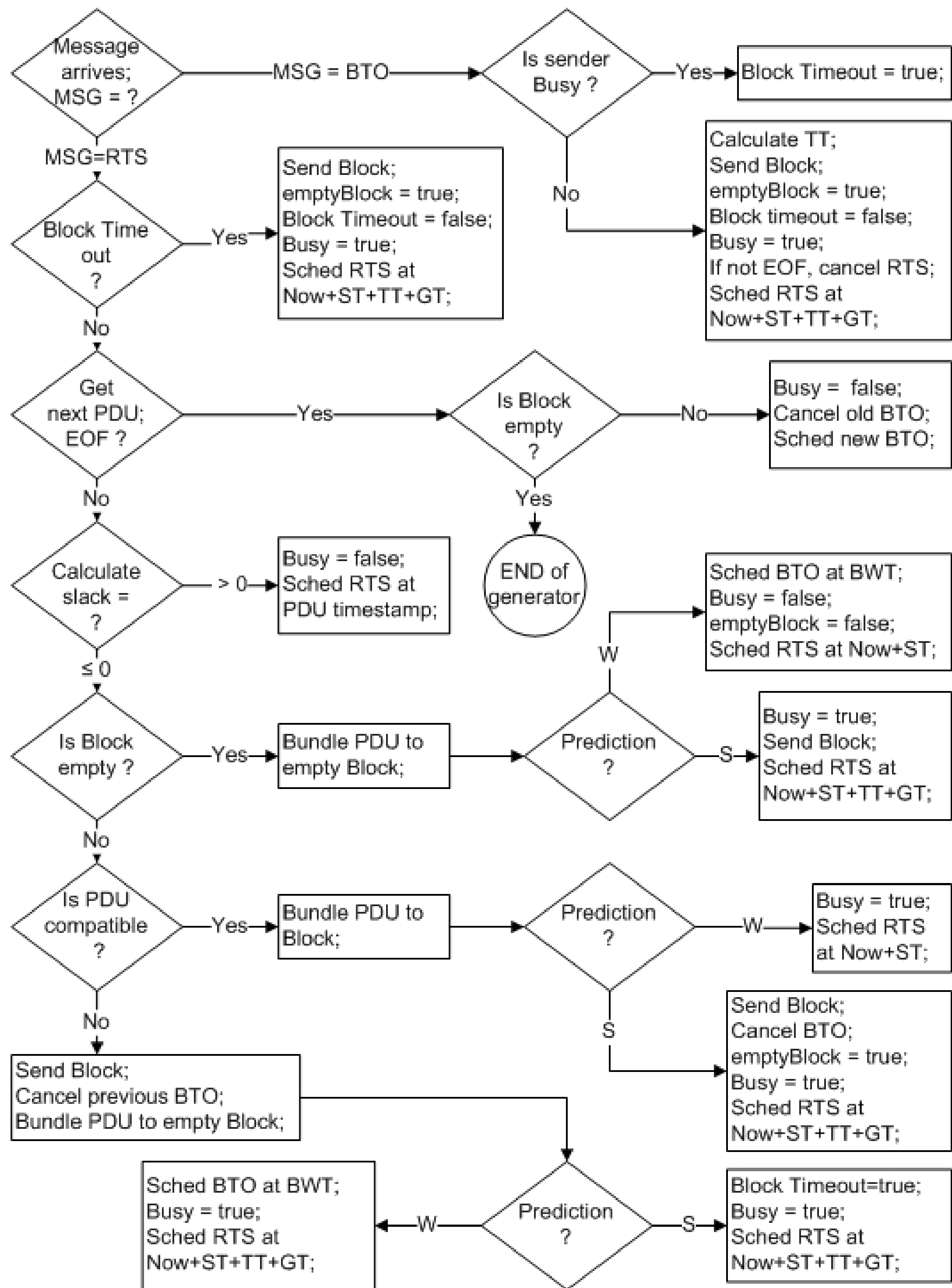


Figure 18: Decision Tree of the Algorithm Used by Generators for Sending PDUs to the Network

be sent immediately, but if it is busy transmitting an older packet, then the current bundle has to wait for the end of that transmission plus the gap time.

ReadyToSend (`RTS`): indicates that the generator is not transmitting and is ready to send a block that has timed out, or it is ready to get the next PDU from the summary file. During reading the next PDU, if the EOF condition arises then the current block of bundled PDUs is examined. If the block is empty, then the generator stops, otherwise it sets the conditions to proceed to send the bundle.

If a new PDU is successfully retrieved form the summary file, then the generator calculates its slack time. If the slack is positive, then the generator schedules an `RTS` to be awaken at the PDU timestamp. If the slack is negative, then the generator is behind the schedule and proceeds to bundle the PDU to previous PDUs already bundled, or to initiate a new bundle, or to send the current bundle which is not compatible with this PDU and create a new bundle starting with this PDU. In any case, the prediction character (`W` or `S`) is read and the generator acts accordingly, either keeping the bundle in case of `W`, or sending it in case of `S`.

If a bundling operation is performed, some time should be spent in the bundle operation itself. In other words, it is not possible to process a PDU (read it and bundle it) in zero time. Each *schedule ready-to-send* operation (`Sched RTS`) includes some minimum time (5 microseconds) for the bundle operation. This time could be considered as generator *service time* (`ST`). The busy time is then computed as the transmission time (`TT`), plus the gap time (`GT`), plus any generator service time (`ST`) if a bundling operation is carried out.

In this Chapter we proposed the lossless *Packet Alloying* technique, in which DIS PDU packets are bundled together prior to transmission based on PDU type,

internal structure, and content. At the receiving nodes, the packets are replicated as necessary to reconstruct the original packet stream. The proposal of the alloying algorithm involves decisions about to bundle or not to bundle two consecutive PDUs. Possible actions are bundling the incoming packet to the current block, or sending the current block and start a new block. Three variables are considered in this algorithm: the *type*, the *length*, and the *timestamp* of each PDU. Observations taken from PDUs participating in negative spikes indicate that if the type and the length of consecutive PDUs are the same, they are compatible and can be bundled according to the definition given in Section 4.3.1.

Chapter 5 studies several simulation vignettes using heuristic parameters, but not applying Packet Alloying bundling. Chapter 5.8 analyzes bundling strategies including *Always-Wait*, *Always-Send*, *Neural-Network*, *Type*, *Type-Length*, and *Type-Length-Time*, and the results of the bundling strategies are compared to those obtained without bundling.

CHAPTER 5

EMBEDDED SIMULATION TRAFFIC ANALYSIS

In this Chapter, different bandwidth capacities are simulated and analyzed, using OTB logs from three different vignettes. PDU travel time and slack time, router and satellite queue lengths, and number of packet collisions are assessed at 64 Kbps, 256 Kbps, 512 Kbps, and 1 Mbps capacities. Results indicate that a *Type-Length* offline prediction strategy is capable of reducing travel time by an upper bound of 85 %, slack time up to 97 %, queue length up to 98 % on bandwidth-restricted channels of 64 Kbps.

5.1 Processing Flow and Sequencing

The simulator was run on data collected from several vignettes. Initially, a vignette developed by Hubert Bahr for his Ph.D. research was used [Bah04]. This vignette contained PDUs sent by one single entity to test the proof of concept. A second simulation was run on a vignette containing two senders. Because of their simplicity, these vignettes were used mainly to test the simulator and to have some insight about the preprocessing that had to be performed on the input data prior to the simulation phase.

The most important outcomes in this research result from the MR1 vignette described in Appendix A, which includes 6 senders from a total of 24 vehicles. Four sets of simulations were performed on it, varying the assignment of senders to computer nodes. Additionally, simulations were run to observe the effect of applying PDU bundling, as well as the usage of the Head-of-Line (HoL) priority in router queues.

5.2 Input Data and AWK preprocessing

The input data to the model is obtained from the OTB logger. After setting up a particular vignette, the OTB logger records all the PDUs generated into an output file that is later converted to ASCII text. OTB generates ESPDUs and Persistent Object PDUs (PO_PDUs) that are a subclass of the general category of PDUs. PDU formats are defined in the IEEE Standards [IEE95a], [IEE95b], [IEE96] and [IEE98]. Although PO_PDUs have a somewhat different format as compared to IEEE PDUs, a considerable degree of similarity exists that allows the application of the IEEE standards to PO_PDUs in order to extract information about the type, length, and timestamp of each message.

The raw data collected by the OTB logger is not directly used as input to the simulator due to the large size of the file (265 MBytes for the MR1 vignette) and to the amount of data stored therein, which are unnecessary for the communication traffic analysis. The log is an ASCII file containing the descriptions of all the transmitted PO_PDUs. Figures 21 and 22 display samples of the raw log.

When OTB ends processing the vignette, an `awk` program (see Appendix C) parses the ASCII file and collects only those PDU variables required during the simulation (type, length and timestamp) into a summary PDU file, along with two

newly created counters. One counter represents a local PDU ID for the generating site, and the other is a global ID from among all the participating sites.

The OTB logger does not save PDUs in strict ascending order of timestamp within the input files. In the logged files collected from running the vignettes there were found sequences of 40 or more PDUs bearing exactly the same timestamp, as well as cases of PDUs stored in reverse chronological order. Due to these anomalies between when the PDU was timestamped and its contents are logged, the PDU files are sorted chronologically previous to running the `awk` program that assigns the local and global IDs, keeping the original relative order for records having the same timestamp. Figure 19 depicts a general view of the steps involved in the simulation process.

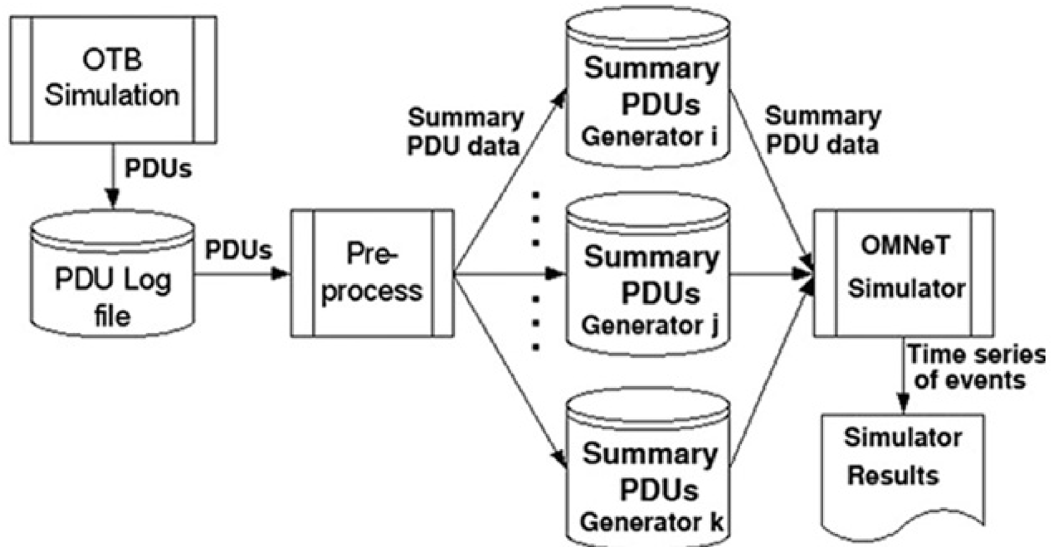


Figure 19: Overview of the Simulation Process

The `awk` parser creates a separate file for each different transmitting site found, and names it `data`*n*`.txt`, where n is the site ID. Each `data`*n*`.txt` file contains summaries of all the PDUs generated by site n. Later, during the simulation each

generator n reads in the corresponding PDU summary file `datan.txt` created out of OTB logged data. A sample of the resulting files `datan.txt` is shown in Figure 20.

Hex Timestamp	Length (Bytes)	Timestamp mm:ss	Global PDU ID	PDU type	Local PDU ID	Prediction
0x4f690c7a	32	18:36.707	1	acknowledge	41	S W W S
0x4f7ab058	32	18:37.676	2	acknowledge	73	S W W S
0x4f8ca818	32	18:38.663	3	acknowledge	112	S S S S
0x4ffc8a88	92	18:44.809	4	po_simulator_present	310	S S S S
0x513da63a	100	19:02.448	5	po_objects_present	977	S S S S
0x51752798	92	19:05.497	6	po_simulator_present	1084	S W W S
0x531629f4	92	19:28.404	7	po_simulator_present	1687	S S S S
0x53617074	100	19:32.539	8	po_objects_present	1868	S S S S
0x548db8ba	92	19:49.034	9	po_simulator_present	2365	S S S S
0x558cfbfa	100	20:03.056	10	po_objects_present	2805	S S S S
0x55fe2df6	92	20:09.274	11	po_simulator_present	3000	S W W S
0x57a315a2	92	20:32.395	12	po_simulator_present	3717	S S S S
0x57b565ee	100	20:33.401	13	po_objects_present	3768	S S S S
0x5917877a	92	20:52.854	14	po_simulator_present	4393	S S S S
0x59e1a736	100	21:03.957	15	po_objects_present	4721	S S S S
0x5a8738fc	92	21:13.052	16	po_simulator_present	5039	S W W S
0x5c357768	92	21:36.686	17	po_simulator_present	5728	S S S S
0x5c357768	100	21:36.686	18	po_objects_present	5729	S S S S
0x5ca513f0	84	21:42.817	19	po_point	5972	W W W W
0x5ca513f0	84	21:42.817	20	po_point	5973	W W W W
0x5ca513f0	84	21:42.817	21	po_point	5974	W W W W
0x5ca513f0	84	21:42.817	22	po_point	5975	W S S S

Figure 20: Sample of the Contents of Summary File `Datan.txt`

The information listed in Figure 20 is interpreted as follows from left to right: hexadecimal timestamp, PDU length in bytes, separator "—", decimal timestamp, local PDU ID equivalent to the current position of the PDU within the file, PDU type surrounded by angle brackets, global PDU ID, and the four letters `S` and `W` already explained in Section 4.3.2.2.

In the DIS protocol, timestamps are stored in 32 bits. In order to convert the timestamps into decimal format, each unit of the unsigned integer value stored

in the leftmost 31 bits represents $1/(2^{31}-1)$ of an hour, giving the timestamp a resolution of less than 0.5 nanoseconds. The rightmost bit of the timestamp is a flag that indicates whether the time is relative to an arbitrary initial time, or absolute UTC time. For example, the value in the hex timestamp `0x4f690c7a` corresponds to $4F690C7A_{16}/2 = 27B4863D_{16} = 666142269_{10}$ units in decimal, and $666142269_{10}/(2^{31}-1) = 0.31019666$ hours or 18:36.707, as indicated in the starting time of the sample file in Figure 20.

PDU lengths vary widely, as seen in the sample of Figure 20. In the studied vignettes, the minimum PDU length found is 26 bytes, the maximum is 1,368, with an average of 211 bytes. The sample also shows the last four PDUs bearing the same timestamp and type, which makes them compatible for bundling as discussed later. The contents shown in Figure 20 range through 21:42.817 min:sec.millisec in the simulation time clock for a single generating source which transmitted four different types of PDUs each with different contents.

5.2.1 PDU Example 1

As a first example of a complete PDU, Figure 21 lists the ASCII equivalent of the fields in a `po_variable` PDU. From among all the fields, the most important to the simulator are those containing the transmitting site identification, the length in bytes and the timestamp. The example shows `do_header.object_id.simulator.site = 1082`, `po_header.length = 147` bytes, and `dis_header.timestamp = :01:33.432` (1 minute, 33 seconds and 432 milliseconds). The timestamp represents the time the entity generated this PDU and enqueued it on the output queue. It could be *absolute* or *relative*, where absolute means that the time is synchronized to Universal

Coordinated Time (UTC) time, and relative means that the clock is not synchronized to UTC, but to any arbitrary time.

The first line in the PDU contains the type, followed by the DIS header fields totaling 12 bytes. The hexadecimal number in the second column represents the address of each field. The other fields have no open documentation, but the OMNeT simulator does not require them for traffic analysis. Obviously, the actual PDUs transmitted by OTB are not in ASCII code, but in binary format. This research had no access to the binary data because details related to PO_PDUs were restricted. Therefore, relevant internal aspects of the PO_PDU format are treated as being similar to the PDUs described in the IEEE standards. For example, the type of PDU displayed in this example is `po_variable`, as seen in its first line. It is not displayed in the usual format `field = value`, but the IEEE standard 1278.1 [IEE95a] indicates that the binary format is prefixed by a PDU header containing its type, among other variables, and so it was assumed by the `awk` parser that this first line represents a field in the header containing the PDU type.

Other assumptions include the length and format of some variable fields, which are undocumented in the IEEE standards and whose length in ASCII characters differs from their length in hexadecimal representation. A C++ parser written specifically to compare PDUs and calculate their differences, listed in Appendix C, assumed the hexadecimal length in this situation.

5.2.2 PDU Example 2

The second example of a short PDU is shown in Figure 22. As before, from the entire PDU only the parameters `originating_id.site = 1086`, `header.sizeof = 32`, and `header.timestamp = :00:35.003` are extracted by the `awk` parser and

```
                                            Hex.       Value
Field Name                                Address    Dec.  Hex.
----------------------------------------------------------------------
<dis204 po_variable PDU>:
dis_header.version                      = 853cfb0 =     4 = 0x04
dis_header.exercise                     = 853cfb1 =     1 = 0x01
dis_header.kind                         = 853cfb2 =   250 = 0xfa
dis_header.family                       = 853cfb3 =   140 = 0x8c
dis_header.timestamp                  = 0x6a4e556 = :01:33.432 (rel.)
dis_header.sizeof                       = 853cfb8 =   196 = 0x00c4
po_header.po_version                    = 853cfbc =    28 = 0x1c
po_header.po_kind                       = 853cfbd =     2 = 0x02
po_header.exercise_id                   = 853cfbe =     1 = 0x01
po_header.database_id                   = 853cfbf =     1 = 0x01
po_header.length                        = 853cfc0 =   147 = 0x0093
po_header.pdu_count                     = 853cfc2 =  7905 = 0x1ee1
do_header.database_sequence_number      = 853cfc4 =     0 = 0x00000000
do_header.object_id.simulator.site      = 853cfc8 =  1082 = 0x043a
do_header.object_id.simulator.host      = 853cfca = 23825 = 0x5d11
do_header.object_id.object              = 853cfcc =   685 = 0x02ad
do_header.world_state_id.simulator.site = 853cfce =     0 = 0x0000
do_header.world_state_id.simulator.host = 853cfd0 =     0 = 0x0000
do_header.world_state_id.object         = 853cfd2 =     0 = 0x0000
do_header.owner.site                    = 853cfd4 =  1082 = 0x043a
do_header.owner.host                    = 853cfd6 = 23825 = 0x5d11
do_header.sequence_number               = 853cfd8 =     1 = 0x0001
do_header.class                         = 853cfda =    11 = 0x0b
do_header.missing_from_world_state      = 853cfdb =     0 = 0x00000000
reserved9                               = 853cfdc =     0 = 0x00000000
variable.total_length                   = 853cfe0 =   132 = 0x00000084
variable.expanded_length                = 853cfe4 =  7812 = 0x00001e84
variable.offset                         = 853cfe8 =     0 = 0x00000000
variable.length                         = 853cfec =   132 = 0x0084
variable.obj_class                      = 853cfee =     8 = 0x08
variable.data                           = "Mine Pallet US M75"
```

Figure 21: Complete PDU of Type Po_variable

stored in the summary file. The receiving site, application, entity, etc. information is not needed to model the PDU traffic. As an observation, this is an acknowledge PDU, which does not contain the `length` keyword, but the `sizeof` keyword instead. For instance, the entries in the summary file corresponding to the PDUs shown in Figures 21 and 22 are:

```
1. 0x6a4e556 147 | :01:33.432   1 <dis204 po_variable PDU>: 1
2. 0x27d3a76  32 | :00:35.003 122 <dis204 acknowledge PDU>: 344
```

```
<dis204 acknowledge PDU>:
header.version               = 8576c78 =  4 = 0x04
header.exercise              = 8576c79 =  1 = 0x01
header.kind                  = 8576c7a = 15 = 0x0f
header.family                = 8576c7b =  5 = 0x05
header.timestamp           = 0x27d3a76 = :00:35.003 (relative)
header.sizeof                = 8576c80 = 32 = 0x0020
originating_id.site          = 8576c84 =  1086 = 0x043e
originating_id.application = 8576c86 = 23825 = 0x5d11
originating_id.entity        = 8576c88 = 65535 = 0xffff
receiving_id.site            = 8576c8a =  1086 = 0x043e
receiving_id.application     = 8576c8c = 23825 = 0x5d11
receiving_id.entity          = 8576c8e = 65535 = 0xffff
ack_flag                     = 8576c90 = 3 = 0x0003
response_flag                = 8576c92 = 0 = 0x0000
request_id                   = 8576c94 = 3 = 0x00000003
```

Figure 22: Short PDU of Type *Acknowledge*

5.3 Transmission Parameters Analyzed

For each experiment, two types of analyses were performed. The first one can be called *independent* or *offline* because it takes place before running the simulator. The second one corresponds to the results obtained by running the OMNeT discrete event simulator when driven by the corresponding OTB packet logs.

5.3.1 Architecture-Independent Bandwidth Analysis

This analysis is subdivided into 2 categories: distribution and assignment of PDUs, and the minimum bandwidth requirement as discussed below.

5.3.1.1 Distribution and Assignment of PDUs

This is a frequency distribution of all the types of PDUs involved in the experiment, as well as the assignment of CPU sites mentioned in the PDUs to simulated generators in computer nodes. The distribution gives information about the sites with more activity that can be taken into account for a better strategic assignment to the available processing nodes.

5.3.1.2 Minimum Bandwidth Requirements

An `awk` program was written to merge the PDUs of all the sites in one single stream of data sorted according to the timestamps. Next, another program calculates the required bandwidth at specific time intervals (2 seconds) without performing any simulation. Computing the bandwidth based on a single data stream is justified by the fact that all PDUs are broadcasted, and so any listening site will have to receive the PDUs from all the generating sites as one single stream of data. Due to traffic changes over time, different minimum *instantaneous* bandwidths over time are required. Instantaneous bandwidth is computed as the ratio of volume of data transmitted to the time interval allotted among consecutive PDUs. No overheads like retransmissions, packet losses, or collisions are considered in the calculation of the bandwidth. However, a time gap separation of 50 microseconds

between consecutive PDUs was included in accordance with the IEEE Std. 802.11 [IEE97]. Therefore, the resulting minimum bandwidth should be seen as an absolute lower bound for the actual bandwidth. The `awk` script in Section C.2 of Appendix C calculates the bandwidth for each set of PDUs. The formal definition of the minimum local bandwidth concept follows.

Definition 5.1: ***Minimum Local Bandwidth***.

> Let all the PDUs in the simulation be sorted in ascending order of timestamp and numbered $\mathrm{PDU}_1, \ldots, \mathrm{PDU}_n$. Let L_i and T_i represent the length in bytes and the timestamp in seconds of PDU_i for all $1 \leq i \leq n$. Let g denote the minimum separating time gap between PDUs in seconds. If $i < j$, the *minimum local bandwidth* for the time interval $[T_i, T_j)$ is the minimum bandwidth in the output channel such that all the consecutive PDUs and gaps $[\mathrm{PDU}_i, g, \ldots, \mathrm{PDU}_{j-1}, g]$ can be successfully transmitted during this interval on or after the times T_k, respectively, for $i \leq k \leq j-1$.

According to this definition, it is allowable to transmit PDU_k on or after the time T_k, provided that at time T_j when the interval has elapsed, all the preceding PDUs have been transmitted. It should be noted that the time interval $[T_i - T_j)$ includes PDU_i and does not include PDU_j. Definition 5.1 assumes that the timestamps are different for consecutive PDUs, such that $T_i \neq T_j$ for all $i \neq j$. If consecutive PDUs bear the same timestamp, the minimum local bandwidth for the corresponding time interval would be infinite, and the PDU sequence is said to be *not feasible*. If the time interval tends to be small relative to the simulation time, then the minimum local bandwidth approaches minimum instantaneous bandwidth, as it is in the extreme case that $j = i + 1$ and only one PDU lies in the interval.

Definition 5.2: ***Minimum Instantaneous Bandwidth***.

The minimum local bandwidth is called *minimum instantaneous bandwidth* if $j = i + 1$ such that the time interval over which it is calculated contains a single PDU.

In practice, any time interval considered short within the context of the simulation time can lead to the concept of minimum instantaneous bandwidth. The successful transmission of PDUs mentioned in definition 5.1 depends on both the length and the timestamp of all the PDUs. The minimum average bandwidth defined next is an approximation to the minimum local bandwidth that in practice gives sufficiently precise results. This measure was used in the architecture-independent analysis of the OTB vignettes.

Definition 5.3: ***Minimum Average Bandwidth***.

Under the same notation and conditions of the definition of minimum local bandwidth, but disregarding the intermediate timestamps $T_{i+1}, \ldots,$ T_{j-1}, the *minimum average bandwidth* is the minimum bandwidth in the output channel such that all the bit volume plus all the gaps separating participating PDUs can be transmitted during the time interval $[T_i, T_j)$.

In calculating the minimum average bandwidth, it is convenient to select a fixed size of S seconds for the time intervals, and divide the total simulation time into subintervals of this fixed size. In such a case, the above definitions can be applied to the time intervals $[T_i, T_j)$, provided that T_i and T_j are separated by a time distance of at least S seconds, but T_i and T_{j-1} are not. In other words, the time interval $[T_i, T_j)$ is of minimum length not less than S seconds, for a given constant S selected a-priori. In the independent analysis of the said vignettes, S was chosen equal to 2 seconds, and so the calculated bandwidth was considered instantaneous.

Theorem 5.1

Under the same notation and conditions of the definition of minimum local bandwidth, and assuming the following condition of feasibility:

$$T_j - T_k - (j-k)g \quad > 0, \quad \forall\ k:\ i \le k < j \tag{5.1}$$

then, the minimum average bandwidth $\overline{B}_{i,j}$ in bits/second required to transmit the PDUs PDU_i, ..., PDU_{j-1} during the time subinterval $[T_i, T_j)$ is calculated as:

$$\overline{B}_{i,j} = \frac{\sum_{k=i}^{j-1} 8L_k}{T_j - T_i - (j-i)g} \tag{5.2}$$

Proof:

The total number of bytes in the PDUs belonging to the interval $[T_i, T_j)$ is calculated as the sum $L_i + \ldots + L_{j-1}$, and the bit volume is 8 times the number of bytes: bits $= \sum_{k=i}^{j-1} 8L_k$. The time allotted to transmit those bits is the interval length $T_j - T_i$ minus each one of the time gaps g that follows every PDU, which leaves a net time of $T_j - T_i - (j-i)g$ because there are $j - i$ gaps. Equation 5.1 is then the quotient of the total number of bits transmitted over the available time to do so.

The condition of feasibility in Equation 5.1 says that the remaining time from T_i to T_j should be sufficient to accommodate all the gaps between PDUs and still have capacity for the transmission of content bytes. The average bandwidth for the interval $[T_i - T_b)$ is the ratio of the total number of bits transmitted over the remaining time in the interval once the gaps have been deducted, as stated in Equation 5.2.

Theorem 5.2

Under the same notation and conditions of the definition of minimum local bandwidth, and assuming the condition of feasibility given in theorem 5.1, if $a < b$ and PDU_a, …, PDU_b are consecutive PDUs in the sequence, then the minimum local bandwidth $B_{a,b}$ for the interval $[T_a, T_b)$ is given by:

$$B_{a,b} = \max_{a \leq k < b} \{\overline{B}_{k,b}\} \tag{5.3}$$

Proof:

If only one PDU lies in the interval, say PDU_{b-1}, then the average bandwidth $\overline{B}_{b-1,b}$ is by definition the minimum local instantaneous bandwidth. By induction on the number of PDUs in the interval, if equation 5.3 is true for PDU_i, …, PDU_b, and PDU_{i-1} is added at the beginning of the sequence, then two cases arise:

Case 1: the new average bandwidth $\overline{B}_{i-1,b} \leq \overline{B}_{i,b}$. In this case, PDU_{i-1} contributes to decrease the average bandwidth. By keeping the highest value, PDU_{i-1} will be transmitted before time T_i and the remaining PDUs can be transmitted without being affected by PDU_{i-1}.

Case 2: $\overline{B}_{i-1,b} > \overline{B}_{i,b}$. The old average bandwidth $\overline{B}_{i,b}$ is not sufficient for a successful transmission of all the bits. Incrementing the average to its new value $\overline{B}_{i-1,b}$ delays the actual starting time of PDU_i, …, PDU_b, as PDU_{i-1} will finish transmission past time T_i, but all the PDUs in the sequence will be successfully transmitted at the new higher bandwidth.

In both cases, Equation 5.3 holds.

According to Theorem 5.2, if the bandwidth for the whole interval $[T_a, T_b)$ is constant, it should not be less than any individual average bandwidth $\overline{B}_{i,b}$. Considering that the starting time for transmitting PDUs can be delayed within the interval,

Equation 5.3 takes the maximum of all components $\overline{B}_{i,b}$ representing the minimum bandwidth requirement.

5.3.2 Analysis of Simulation Results

This analysis of the modeled traffic under each protocol is subdivided into 4 categories:

a) Slack Time Analysis. Statistics about the slack time of all the PDUs generated at a particular site are graphed and discussed. The slack time T_{slack} of a given PDU is defined as the difference between its timestamp and the current simulator time at the moment the PDU is read from the summary input file. More formally, if T_{stamp} represents the timestamp of a PDU and T_{read} represents the time when the PDU was read, then

$$T_{slack} = T_{stamp} - T_{read} \tag{5.4}$$

If the difference is positive, as indicated by $T_{slack} > 0$, then the generator is ahead of the planned schedule, otherwise it is behind it. Thus, a negative slack time indicates that the channel bandwidth is not sufficient to transmit the required PDUs without incurring in delays. If several PDUs are scheduled for transmission at the same timestamp, then they will necessarily produce a negative slack, no matter what bandwidth available. However, the greater the bandwidth, the smaller the magnitude of the negative slack incurred.

b) Travel Time Analysis. Statistics about the travel time of all the PDUs collected at a particular sink are graphed and discussed. The travel time is the difference between the sending time of a PDU from a network node generator

and the arrival time at the node sink. All the transmission times, propagation times and waiting times in router queues are part of the travel time. If T_{stamp}, T_{arr} and T_{trav} represent the release time (timestamp), the arrival time and the travel time of a given PDU, then

$$T_{trav} = T_{arr} - T_{stamp} \tag{5.5}$$

c) Queue Length Analysis. There is a message queue at the satellite and another at each router to store incoming PDUs pending service. Each time a PDU arrives at the satellite or a router, the number of other messages in the system is counted, including the PDUs already in the queue, plus any one being serviced. This count value along with the arrival time of the incoming PDU is recorder in an OMNeT statistics file. When the simulation ends, a separate program processes the file and obtains the statistics about the number of messages in that system.

d) Collision Analysis. The satellite and routers keep separate counters of collisions received from each of the links they are connected to. The satellite is connected to the wireless links WSP and WGS and the routers are connected to the LAN, WSP and WPP links as shown in Figure 7. Each time a collision is detected, the corresponding counter is incremented and its new value along with the current simulation time is recorded for future processing.

The slack time analysis at the ground station shows that many negative and positive spikes are present at regular time intervals. The size of such spikes depend on the timestamp and the length of the PDUs, as well as on the bandwidth of the channel. The following definitions and theorem formalize the occurrence and calculations of such spikes.

Definition 5.4: *Busy Generator*.

The generator of PDUs is in the *busy* state at time t if it is transmitting a packet or waiting for the completion of a time gap separator at that time.

Definition 5.5: *Busy Phase*.

Given a sequence $\text{PDU}_1, \ldots, \text{PDU}_n$ of PDUs of lengths $L_1, \ldots, L_n$ bits, respectively, and timestamped at ascending times $T_1, \ldots, T_n$ seconds, respectively, then any subsequence PDU_i, $\text{PDU}_{i+1}, \ldots, \text{PDU}_{i+k}$ constitutes a *busy phase* if the following conditions are true:

1. The generator is not busy at time T_i when PDU_i is released.
2. For $j = 1, 2, \ldots, k$, when PDU_{i+j} is released at time T_{i+j} the generator is busy completing the transmission of previous PDUs.
3. If PDU_{i+k+1} exists, the generator is not busy at time T_{i+k+1}.

Definition 5.5 indicates that the entire sequence of PDUs can be partitioned into disjoint phases of consecutive PDUs, and each PDU belongs to one and only one phase. The following theorem calculates the size of negative slack spikes during a busy phase.

Theorem 5.3

The magnitude of a negative slack spike of a busy phase PDU_i, $\text{PDU}_{i+1}, \ldots, \text{PDU}_{i+k}$ transmitted at a bandwidth B bps with gap intervals of g seconds can be calculated as:

$$m = \max_{0 \leq j \leq k} \{m_{i+j}\} \tag{5.6}$$

where $m_{i+j} = T_{i+j} - \left(T_i + j \cdot g + \frac{\sum_{u=0}^{j-1} L_{i+u}}{B} \right)$

Proof:

As explained in Equation 5.4, the slack time of a PDU is $T_{slack} = T_{stamp} - T_{read}$. The timestamp for PDU_{i+j} is T_{i+j}. It only remains to calculate the time at which PDU_{i+j} is read by the generator. The busy phase started at time T_i. By definition, during a busy phase the generator is either transmitting or waiting for a separating gap, and there are j PDUs before PDU_{i+j}. Therefore, by the time T_{read} when the generator reads PDU_{i+j}, it will have waited for j gaps, totaling $j \cdot g$ seconds and will have transmitted $\sum_{u=0}^{j-1} L_{i+u}$ bits in $(\sum_{u=0}^{j-1} L_{i+u})/B$ seconds. Therefore, PDU_{i+j} contributes with a negative spike of magnitude m_{i+j} as indicated in the theorem. The magnitude for the whole busy phase is calculated as the maximum of the individual negative spikes. This is because the largest value during the busy phase indicates the greatest degree to which the simulator was behind schedule and hence the height of the negative spike.

If a busy phase contains only one PDU, then Theorem 5.3 yields zero for the magnitude of the spike, and yields a strictly positive value for two or more PDUs, accepting by convention that the summation from index 0 to index -1 is zero. In other words, phases consisting of only one PDU produce no negative spike, and phases of two or more PDUs can produce a negative spike.

Positive spikes are always produced at the end of a phase, provided that the next phase does not start exactly at the end of the previous phase, including the gap separators as part of the phase time. In other words, positive spikes are produced by the time interval separating busy phases. The following theorem formalizes the concept.

Theorem 5.4

The magnitude of a positive slack spike between consecutive busy phases (PDU_i, ..., PDU_{i+k}) and (PDU_{i+k+1}, ..., PDU_{i+k+r}) is calculated as:

$$m = T_{i+k+1} - \left(T_i + (k+1) \cdot g + \frac{\sum_{u=0}^{k} L_{i+u}}{B} \right) \tag{5.7}$$

Proof:

There are $k + 1$ PDUs in the first busy phase, totaling $(k + 1) \cdot g$ seconds of waiting gap time. The effective transmission time of all the PDUs in the first phase is $\sum_{u=0}^{k} L_{i+u}/B$. Therefore, the generator will be busy transmitting the first phase until the time $T_i + (k + 1) \cdot g + \sum_{u=0}^{k} L_{i+u}/B$. The difference between this time and the beginning of the next phase at T_{i+k+1} constitutes the positive spike, which represents the time when the generator is not busy.

5.4 Simulation ST: Vignette With Single Transmitter

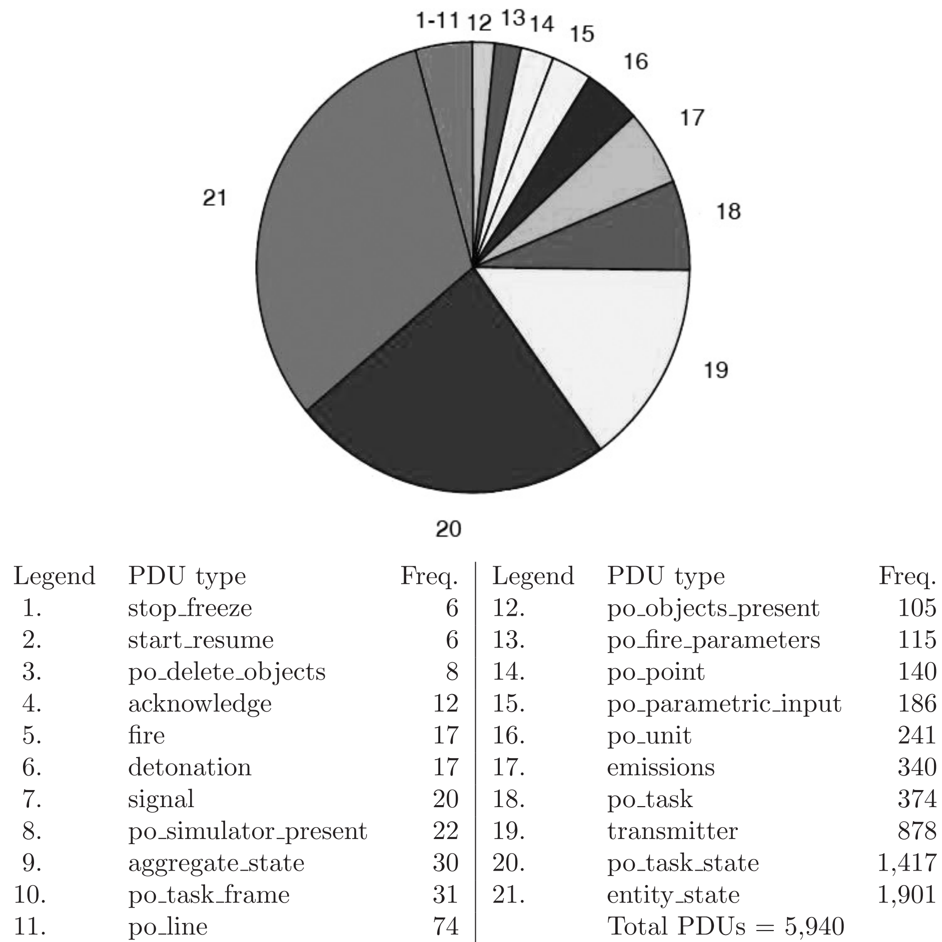
The vignette of this first simulation produced a log file of 28 MBytes of PDU data. The simulation time spanned from 00:14.341 to 07:22.446, for a time period of 7 minutes and 8.105 seconds. A total of 5,940 PDUs were generated by one single site.

5.4.1 Independent Analysis of Logged PDUs

5.4.1.1 Analysis of PDUs and Assignment

Figure 23 shows the distribution and the relative proportion of PDUs according to their types. All the PDUs were generated by the site identified in OTB as site

#1013. During the simulation, this site was assigned to node 0 onboard plane 0. It can be noted that `entity_state`, `po_task_state`, and `transmitter` PDUs are the three most frequent types of PDUs in this simulation, which agree with observations that have been pointed out by several authors (e.g. [Mac95], [SZB96], [BCL97], and [HIL98]).

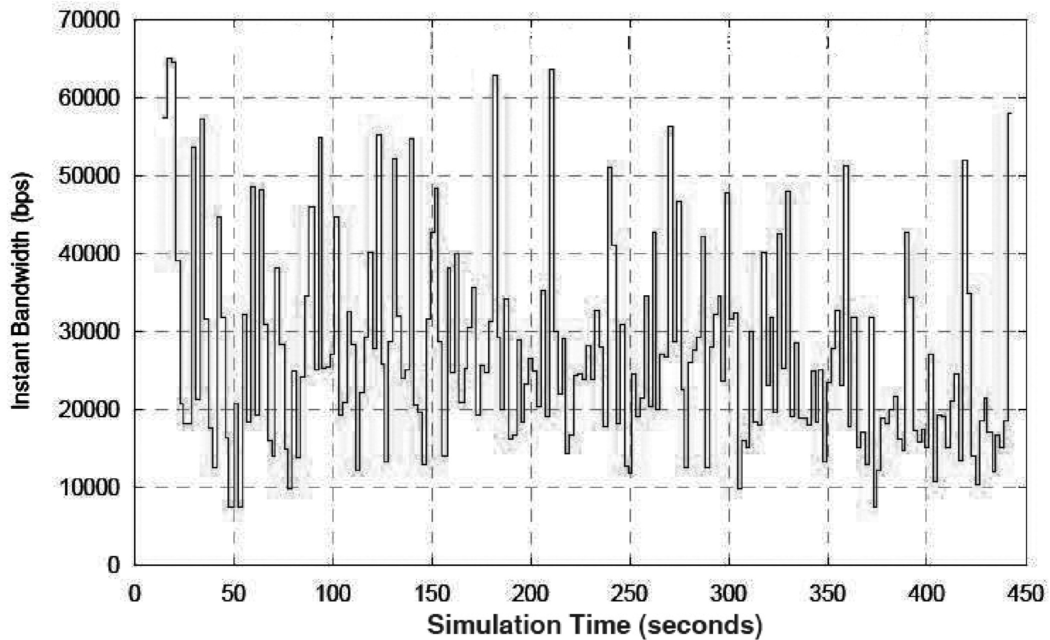


Legend	PDU type	Freq.	Legend	PDU type	Freq.
1.	stop_freeze	6	12.	po_objects_present	105
2.	start_resume	6	13.	po_fire_parameters	115
3.	po_delete_objects	8	14.	po_point	140
4.	acknowledge	12	15.	po_parametric_input	186
5.	fire	17	16.	po_unit	241
6.	detonation	17	17.	emissions	340
7.	signal	20	18.	po_task	374
8.	po_simulator_present	22	19.	transmitter	878
9.	aggregate_state	30	20.	po_task_state	1,417
10.	po_task_frame	31	21.	entity_state	1,901
11.	po_line	74		Total PDUs = 5,940	

Figure 23: PDU Type Distribution Generated in Simulation ST

5.4.1.2 Minimum Bandwidth Requirements

Figure 24 shows the minimum instantaneous bandwidth required at each interval of 2 seconds. The graph is typical of a burst transmission, having instances of heavy traffic followed by others of low usage. According to the results, all the instantaneous bandwidths lie in the range of 7.3 Kbps to 65 Kbps, with an average of 27 Kbps. Therefore, a standard value of 64 Kbps in the wireless channels should be sufficient to handle all the traffic in this simulation.



Samples = 206	Minimum bandwidth = 7337.8
Init time = 14.341	Maximum bandwidth = 65065.5
Final time = 442.446	Average bandwidth = 26907.5
	Std deviation = 12630.6

Figure 24: Minimum Bandwidth Requirements

5.4.2 Slack Time

Figure 25 indicates that negative slack occurrences are not readily visible. There are some, but they are not prominent at the graph scale used. Negative slack occurrences also depend on the node assignment of site #1013. If the site had been assigned to the ground station, probably more instances of negative slack would have appeared. The reason is that the generators onboard planes are directly connected to the Ethernet bus running at 100 Mbps. From the generator point of view, the network is very fast and the generator is almost always ahead of schedule. However, the ground station is directly connected to the slow 64 Kbps wireless channel. That promotes more negative slack occurrences.

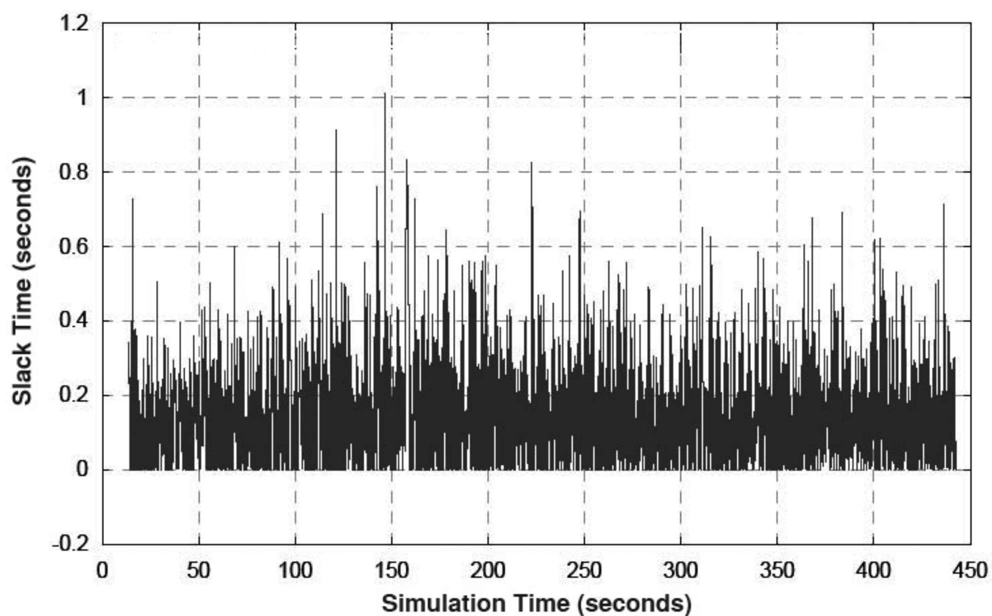


Figure 25: Slack Time to Send Next Message at Generator 0 (64 Kbps)

5.4.3 Travel Time

As depicted in Figure 26, the travel times of PDUs seen by the ground station show that most PDUs took less than 0.6 seconds to arrive at the destination. The minimum travel time is close to 0.255 seconds that correspond to the time needed by a signal to travel from Earth to the satellite and back to Earth. This is 38,300 Km $\times$ 2 divided by the speed of light, yielding 0.255 seconds. The periodicity and spikes observed in Figure 26 will be discussed later.

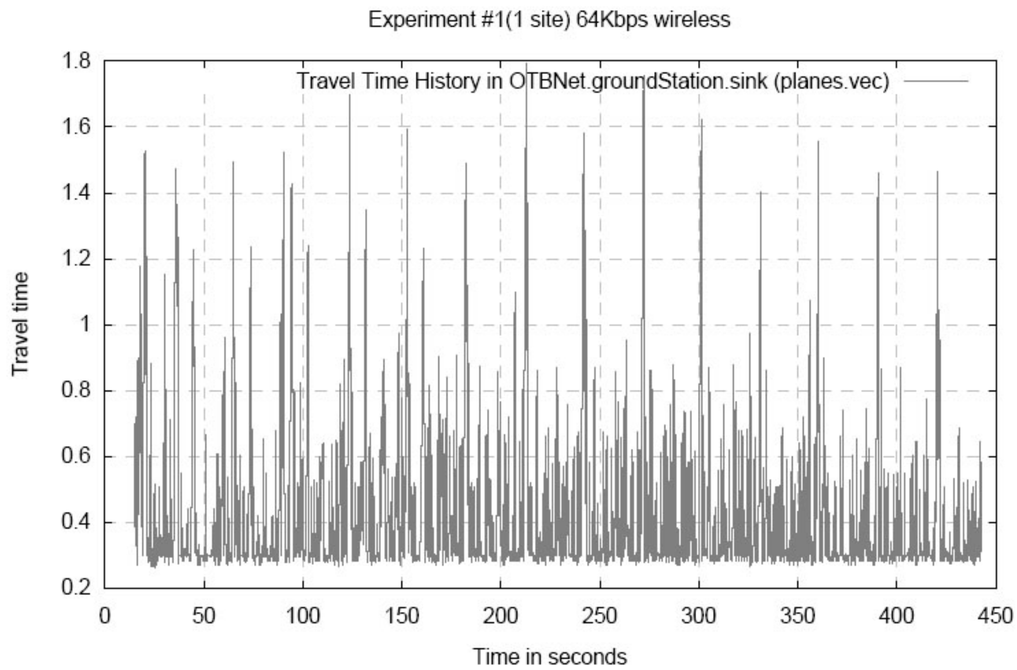


Figure 26: Travel Time as Sensed by Ground Station (64 Kbps)

5.4.4 Queue Length

Figure 27 shows the number of messages at the router of plane 0. The maximum value is less than 45, which is quite acceptable for a router. In Figure 28, the satellite shows even better results with maximum queue less than 16 messages. In both instances, the queue usage has similar characteristics throughout the simulation time, showing peaks of all sizes evenly distributed along the time.

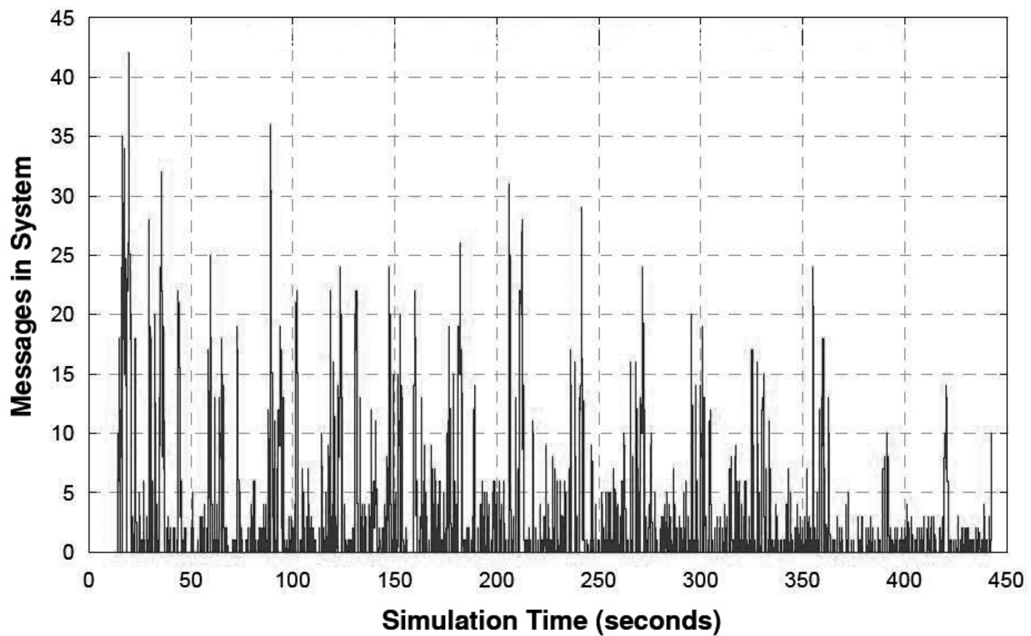


Figure 27: Messages in Router 0 in Plane 0 (64 Kbps)

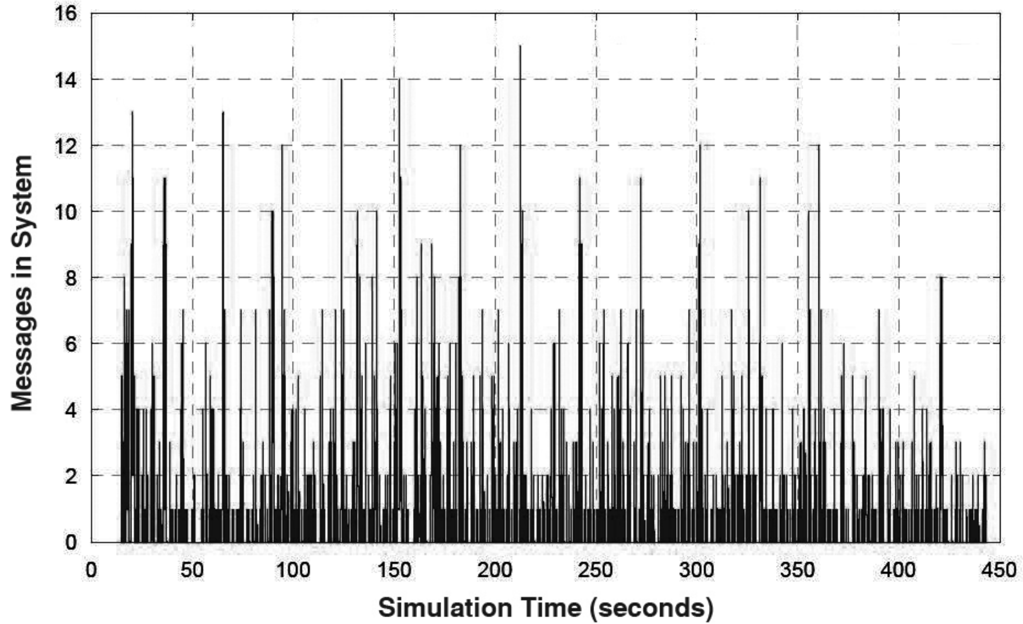


Figure 28: Messages in the Satellite (64 Kbps)

5.4.5 Collisions

No collisions were detected. This is understandable due to the fact that only one site is transmitting. Collisions for multiple transmitting sites will be discussed in experiments that follow.

5.4.6 Conclusions of Simulation ST

This first simulation was based on a single operator vignette to define the quantities using a straightforward example. The main purpose was really to test the simulator itself. All the wireless links were set to an acceptable bandwidth of 64

Kbps. The results obtained are congruent with the expected values for such a simulation based on the time independent analysis.

The behavior of the PDU traffic is typical of that of any computer network using the DIS protocol, as has been reported by other authors. For example, [MZP94] reports graphs of time vs. PDU/second showing similar PDU activity as in Figures 27 and 28, which we will analyze below.

It is interesting to note that the simulator calculates the PDU travel time to the ground station with a lower bound of near 0.25 seconds, based solely on the parameters given, as propagation time of each communication link and distance between sites, which gives another indication of its reliability. As a conclusion, it can be stated that the OMNeT simulator works accordingly with the expected results for this simulation, which gives some degree of confidence in its accuracy. The traffic is perfectly handled at 100 Mbps over the Ethernet link and 64 Kbps over the wireless channels.

From the simulation data, it can be concluded that negative slack at the generator is negligible, the queue lengths in the routers and satellite were less than 45 and 16 messages, respectively, and collisions were not detected. Therefore, 64 Kbps in the wireless channels is sufficient bandwidth for this simulation. It should be noted that the conclusion agrees with the results of the independent analysis, which gives support to the idea that the independent analysis is a valuable tool in a rapid assessment of a network bandwidth.

5.5 Simulation DT: Vignette with Dual Transmitters

The vignette in this second simulation produced a log file of 22 MBytes of PDU data. The simulation time spanned from 00:35.003 to 05:50.574, for a time period

of 5 minutes and 15.571 seconds. A total of 5,430 PDUs were generated by 2 sites identified in OTB as site #1082 and site #1086.

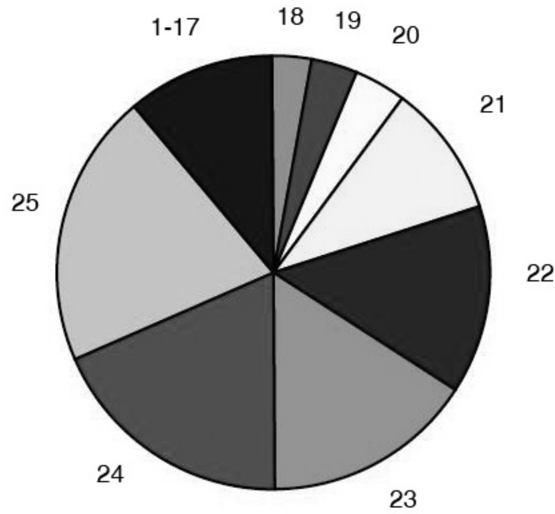
5.5.1 Independent Analysis of Logged PDUs

5.5.1.1 Analysis of PDUs and Assignment

Figure 29 shows the distribution and the relative proportion of PDUs for each type. Sites #1082 and #1086 generated 926 and 4,504 PDUs (17 % and 83 %) respectively, which indicates that in the vignette one site is much more active than the other. During the simulation, site #1082 was assigned to ground station (node 24) and site #1086 was assigned to node 0 onboard plane 0. As in the first simulation, PDUs of types `entity_state` and `po_task_state` are among the most frequently generated.

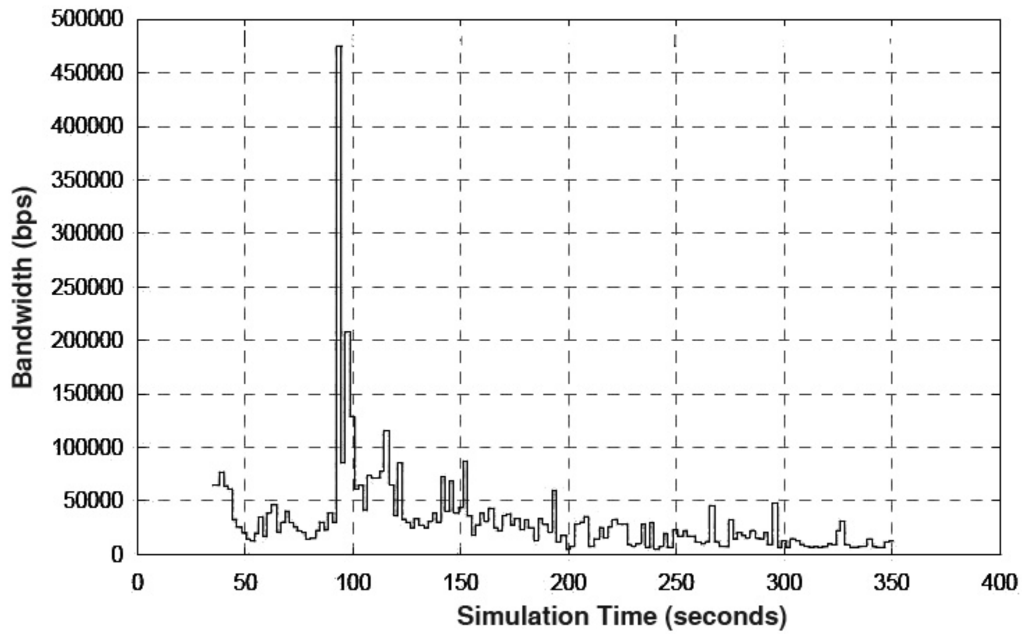
5.5.1.2 Minimum Bandwidth Requirements

Figure 30 shows a high bandwidth spike near second 95. The reason is that many PDUs are scheduled to be sent at times close to second 95. A detailed look at the data shows that during the time interval [92.463, 94.484], 310 PDUs totaling 957 Kbits are being scheduled. Yet, this volume of data requires approximately 474 Kbps to be sent on time and cannot be achieved in over 64 Kbps channel without incurring delay. After second 100, the remaining PDUs can be handled at 64 Kbps, as Figure 30 indicates. Because the bandwidth is considered constant in the actual links, 64 Kbps will be insufficient to fulfill the needs of this second vignette.



Legend	PDU type	Freq.	Legend	PDU type	Freq.
1.	stop_freeze	3	14.	po_line	65
2.	start_resume	3	15.	po_fire_parameters	65
3.	po_delete_objects	7	16.	po_point	80
4.	acknowledge	12	17.	po_task_frame	85
5.	po_link	12	18.	po_parametric_input	165
6.	aggregate_state	18	19.	po_unit	186
7.	fire	24	20.	emissions	204
8.	detonation	24	21.	transmitter	547
9.	po_simulator_present	26	22.	po_variable	760
10.	signal	27	23.	po_task	835
11.	po_parametric_input_holder	40	24.	entity_state	1,021
12.	po_objects_present	52	25.	po_task_state	1,105
13.	po_overlay	64		Total PDUs = 5,430	

Figure 29: PDU Type Distribution Generated in Simulation DT



Init time = 35.003 sec.	Minimum bandwidth = 5153.6 bps
Final time = 350.574 sec.	Maximum bandwidth = 477404.0 bps
Average = 32401.2 bps	Std. deviation = 44953.7 bps

Figure 30: Minimum Bandwidth Requirements

5.5.2 Slack Time

Figure 31 shows the slack time as seen by computer node 0 (line labeled 1) and the ground station (line labeled 2), setting the wireless bandwidth to 64 Kbps. It is clear that in the approximate time interval [90, 120] the ground station suffered from high negative slack occurrences. This is due to the impossibility to handle the data volume at 64 Kbps. After the second 120 the ground station gets recovered from the delay. Figure 32 represents a zoom in of Y axis in Figure 31. Even at this scale, there are no visible negative slack occurrences in plane 0, mainly due to the high bandwidth of the LAN link.

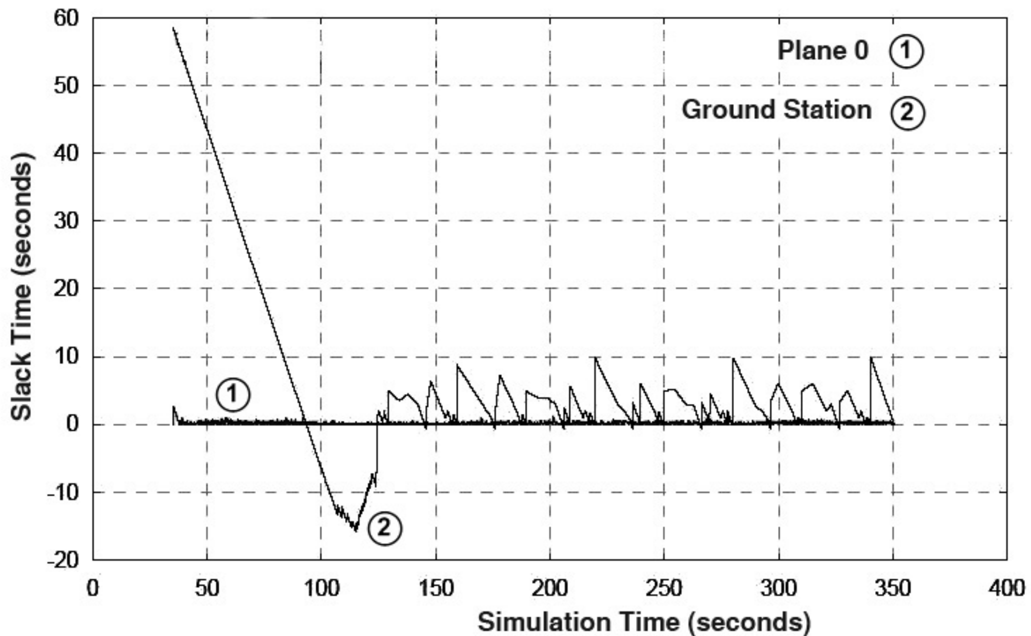


Figure 31: Slack Time to Send Next Message at Plane 0 and Ground Station (64 Kbps)

Figure 33 shows both, the slack at plane 0 (curve numbered 1) and the slack time at the ground station (curve numbered 2) for a wireless bandwidth of 400 Kbps.

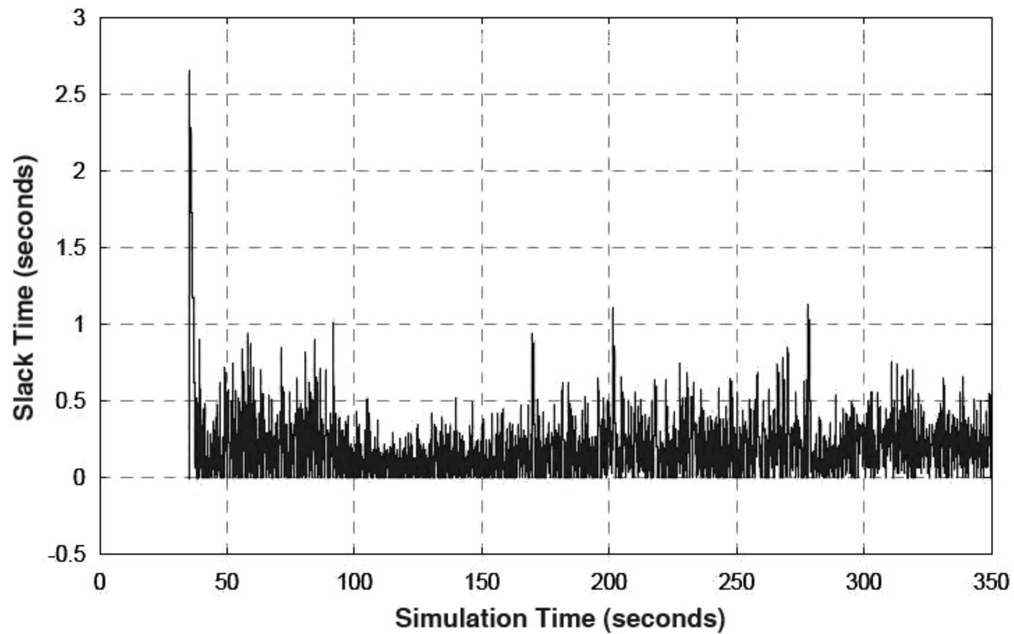


Figure 32: Slack Time to Send Next Message at Plane 0 (64 Kbps)

This non-standard bandwidth was chosen based on the results of the independent analysis. As seen in Figure 33, the slack at the ground station is greatly reduced, but it is still negative before second 100. However, the positive slack occurrences were almost unaffected by the bandwidth increase.

5.5.3 Travel Time

Figure 34 represents the travel time as seen by node 0 (curve labeled 1) and the ground station (curve labeled 2) at 64 Kbps in the wireless channels. Both graphs are quite similar, showing a large delay during the interval from second 90 to second 170.

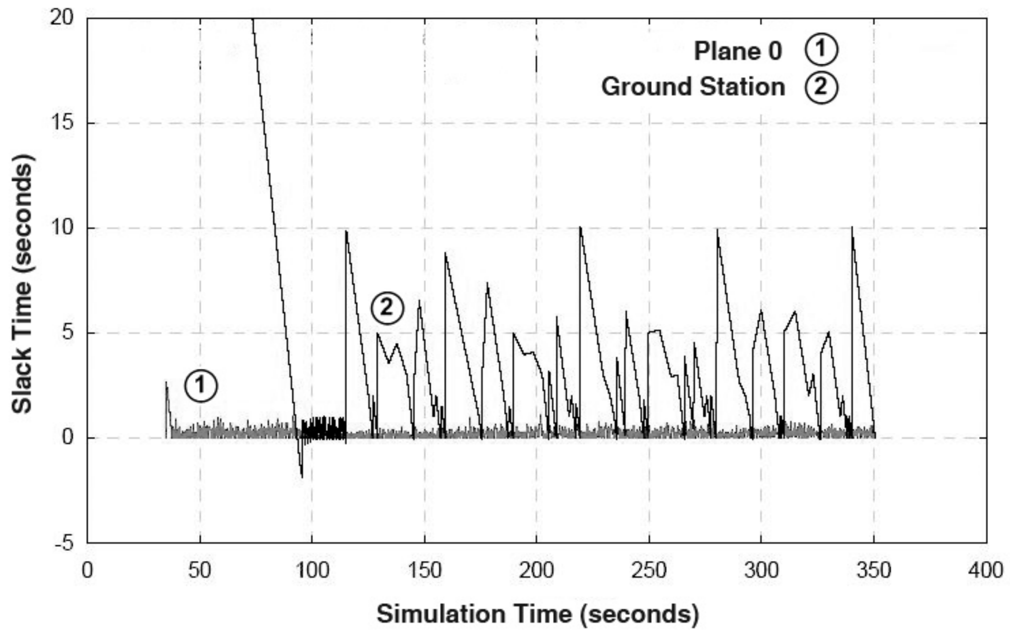


Figure 33: Slack Time for Next Message by Plane 0 and Ground Station (400 Kbps)

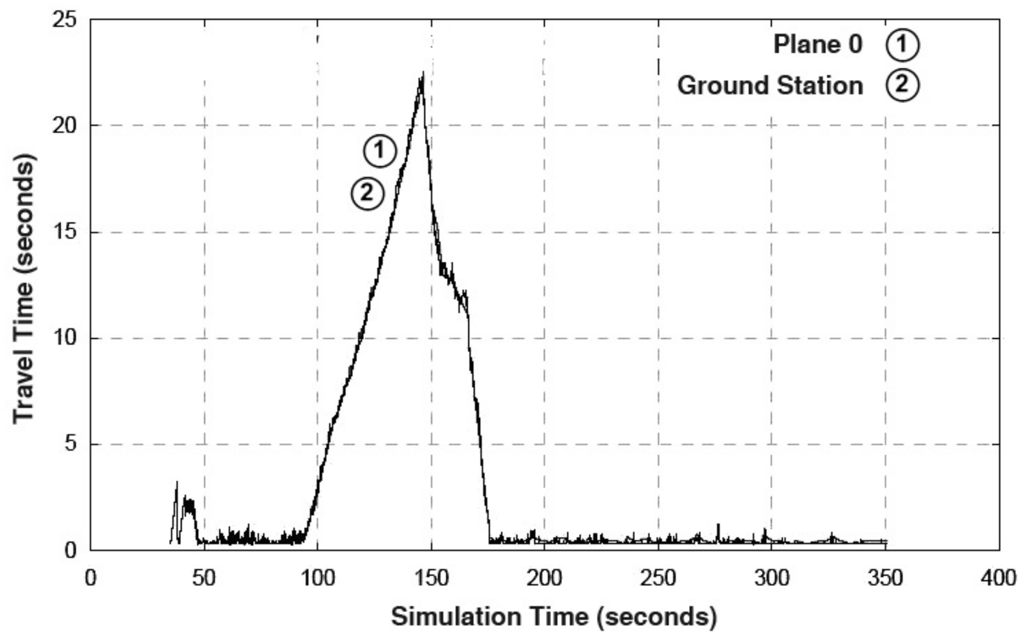


Figure 34: Travel Time at Plane 0 and Ground Station (64 Kbps)

Node 23 located on node 2 of plane 7, produces the graph shown in Figure 35, which was drawn using lines to connect consecutive observations. The graph shows two sets of PDUs. The PDUs originating at the ground station suffer from high delays, while PDUs coming from plane 0 have small delays. The reason is that PDUs coming from plane 0 do not wait at the satellite queue and are not affected by the propagation delay of satellite signals.

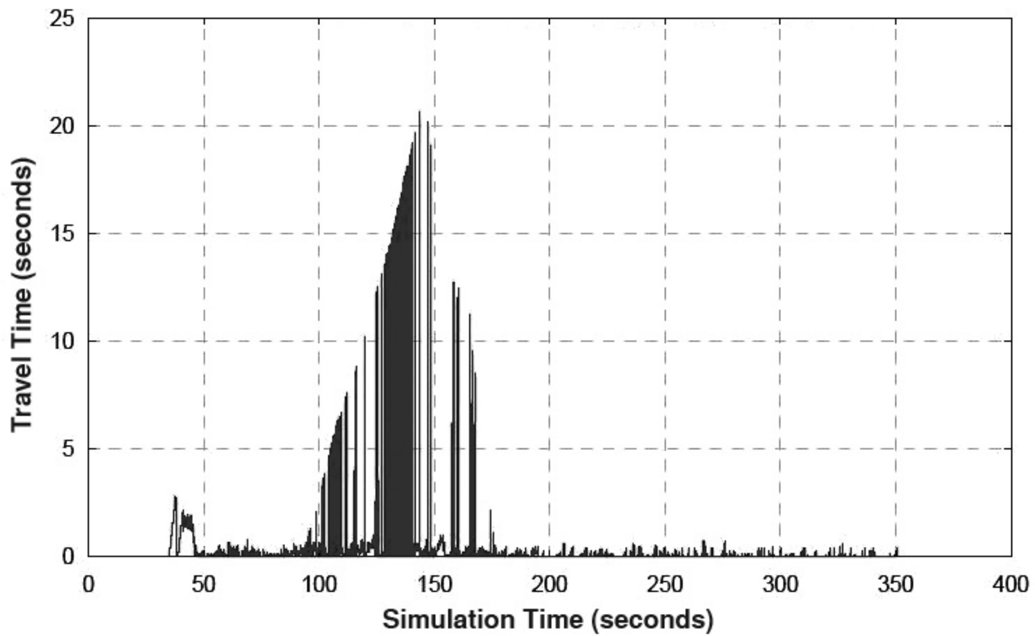


Figure 35: Travel Time at Plane 7 (64 Kbps)

Figure 36 shows travel times seen by node 21 at node 0 on plane 7 when the wireless bandwidth is increased to 400 Kbps. It becomes clear now that this node receives two types of PDUs, being the satellite PDUs delayed by approximate 0.25 more seconds.

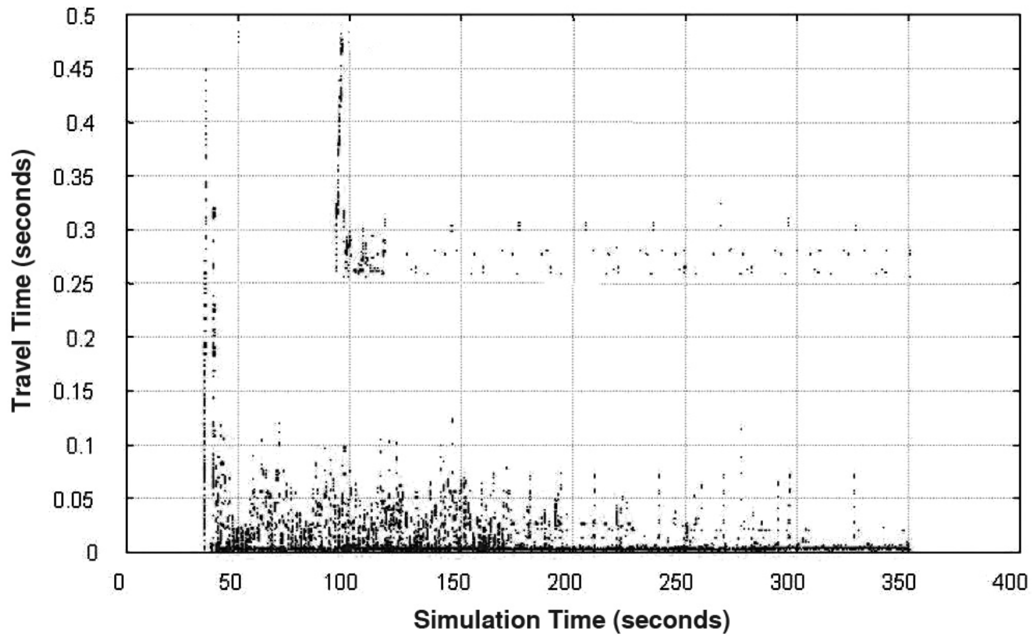


Figure 36: Travel Times at Plane 7 Zoomed In (400 Kbps)

5.5.4 Queue Length

Figure 37 clearly shows that at 64 Kbps in the wireless link, the satellite suffers from a large queue delay during the time interval [100, 170] seconds. This is a strong indication that 64 Kbps are not enough to handle the traffic at the satellite. On the other side, the traffic at the router onboard plane 0 seems capable of handling its corresponding traffic.

Increasing the bandwidth to 400 Kbps greatly improves the throughput of the satellite queue, as indicated in Figure 38. At 400 Kbps, the satellite maintains a queue length fewer than 20 PDUs most of the time. The router onboard plane 0 still shows an initial queue length of 120 messages which does not impact its performance.

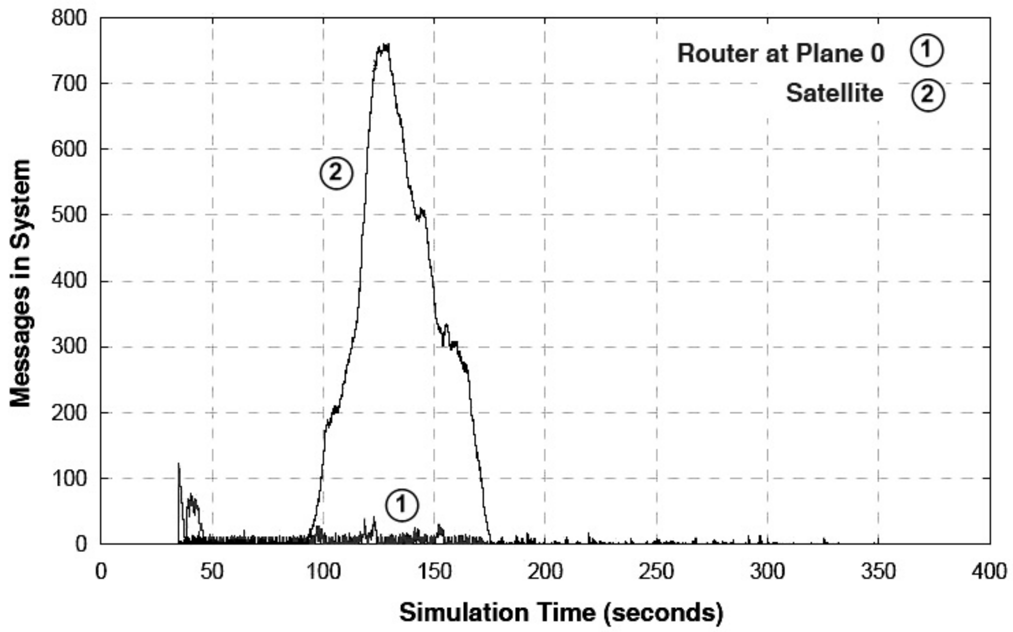


Figure 37: Comparison of Queue Lengths of Plane 0 and Satellite (64 Kbps)

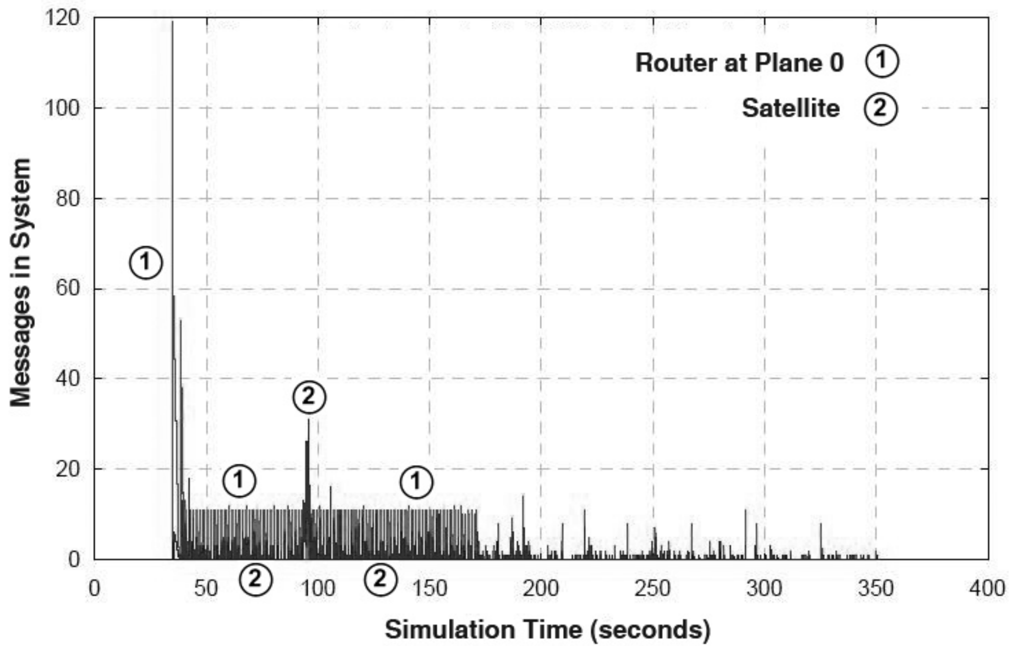


Figure 38: Comparison of Queue Lengths of Plane 0 and satellite (400 Kbps)

5.5.5 Collisions

At 64 Kbps in the wireless channels, Figure 39 shows that some collisions were detected in the WSP channel that connects the satellite to the planes. However, the other channels do not show collision activity. The graph represents the number of collisions detected per second by the router at plane 1.

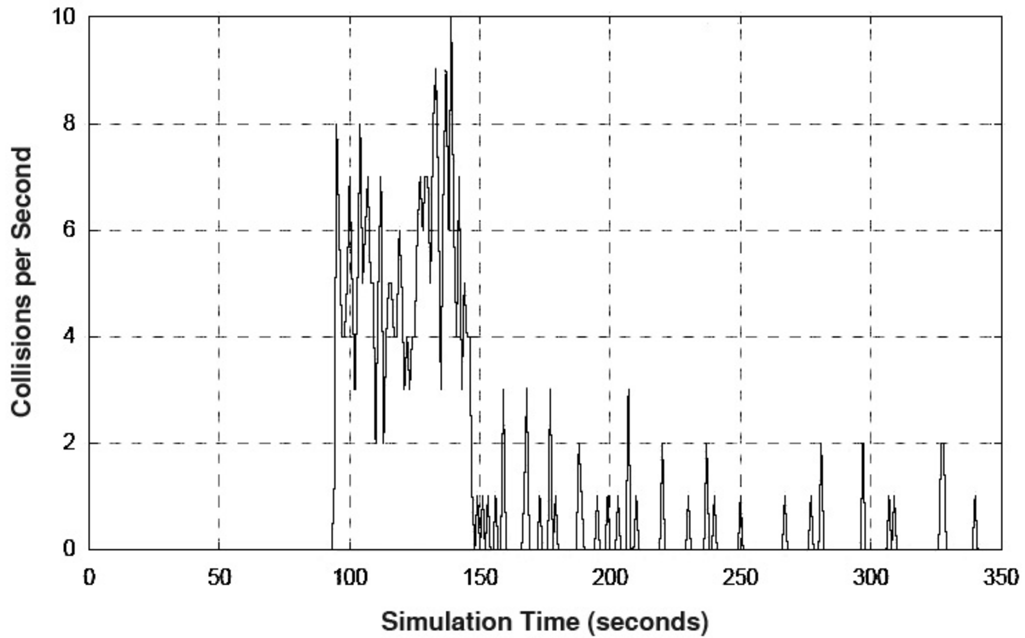


Figure 39: Collisions per Second Detected at Plane 1 (64 Kbps)

Figure 40 gives the number of collisions accumulated along the time, as seen by the router at plane 7. The maximum collision rate occurs in the range [90, 140] seconds, totaling near 280 collisions. Afterwards, the rate evidently decreases and at the end of the experiment the collision counter reaches approximately 325. Considering that the total number of PDUs sent is 5,430, the collisions represent near 6 % of the total number of packets.

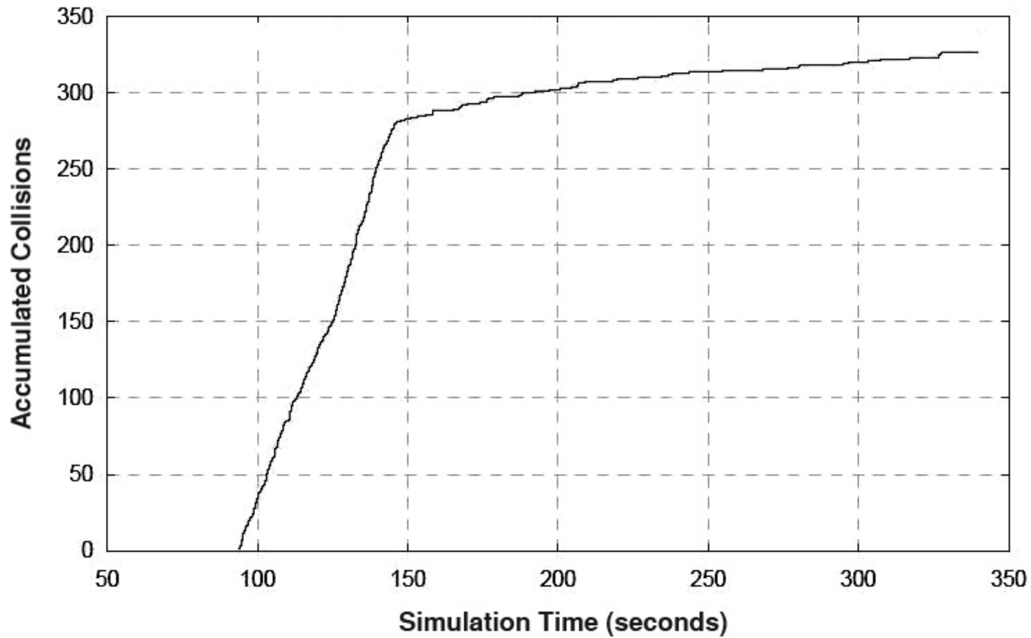


Figure 40: Collision Accumulation Over Time at Plane 7 (64 Kbps)

The current simulator does not include any special retry handling for collisions, like retransmissions using exponential backoff algorithms [Mol94] [IEE97] [FZ02]. However, the small percentage of collisions leads to the conclusion that a more sophisticated simulator including exponential backoff will produce results very similar to the ones produced by the current one.

The simulator was executed with a setting of the wireless channels to 400 Kbps. At this bandwidth, collisions in the WSP link are significantly reduced, totaling fewer than 60 at the end of the experiment. Near second 92, a peak of 8 collisions per second occurs, immediately decreasing to 3 or fewer collisions per second during the rest of the simulation.

5.5.6 Conclusions of Simulation DT

As predicted in the independent analysis, 64 Kbps in the wireless channels is insufficient bandwidth to handle traffic near the interval [90, 175] seconds. After second 175, the traffic becomes less intense and can be handled. Based on the independent analysis, increasing the bandwidth to 400 Kbps in wireless channels produce much better results, with travel times less than 0.5 seconds for all packets. Collisions were detected in the WSP link only, but at 400 Kbps, the total number is less than 60. At 64 Kbps, the queue length was close to 750 PDUs in the satellite, number that decreases under 30 PDUs at 400 Kbps. Assuming the worst case length in the satellite queue, 750 PDUs of 1,368 bytes each would require near 1 MByte of memory, which does not impose an overwhelming demand.

5.6 Simulation MR1T6: Vignette MR1 with Six Transmitters

Simulation MR1T6, as well as the remaining ones, is based on the MR1 vignette described in Appendix A. The log file for this simulation is 265 MBytes long. The simulation time spanned from 17:14.447 to 42:27.808, for a time period of 25 minutes and 13.361 seconds. A total of 60,341 PDUs were generated by 6 sites identified in OTB as site #1519, #1526, #1529, #1532, #1533, and #1538, respectively. The simulation is not using the bundling technique. Bundling simulations are covered in Chapter 5.8.

5.6.1 Independent Analysis of Logged PDUs

5.6.1.1 Analysis of PDUs and Assignment

There are 27 different types of PDUs in the OTB simulation of the MR1 vignette. Figure 4 shows the distribution, the volume of bytes and the relative proportion of PDUs for each type, and Figure 41 depicts the corresponding pie charts according to the labels in the table. The most frequent type of PDU is `entity_state` with 28,569 PDUs (47 %), followed by `po_task_state` with 11,960 PDUs (nearly 20 %).

Table 4: Types of PDUs and Volume of Bytes Transmitted for Each Type

Label	PDU Type	#PDUs	# Bytes	% # PDUs	% # Bytes
1.	laser	3	264	0.005	0.002
2.	start_resume	3	132	0.005	0.001
3.	stop_freeze	3	120	0.005	0.001
4.	po_task_authorization	6	388	0.010	0.003
5.	po_minefield	14	5,384	0.023	0.043
6.	fire	23	2,208	0.038	0.018
7.	detonation	25	2,550	0.041	0.021
8.	acknowledge	36	1,152	0.060	0.009
9.	po_delete_objects	110	4,216	0.182	0.034
10.	minefield	117	42,120	0.194	0.339
11.	po_message	119	69,020	0.197	0.556
12.	signal	237	19,896	0.393	0.160
13.	aggregate_state	256	37,888	0.424	0.305
14.	po_simulator_present	370	34,040	0.613	0.274
15.	po_task_frame	382	87,984	0.633	0.709
16.	mines	386	396,088	0.640	3.19
17.	po_point	659	55,356	1.09	0.45
18.	po_objects_present	682	577,952	1.13	4.65
19.	po_fire_parameters	713	376,464	1.18	3.03
20.	iff	851	51,060	1.41	0.41
21.	po_line	912	115,524	1.51	0.93
22.	po_parametric_input	1196	165,440	1.98	1.33
23.	po_unit	1793	1,161,864	2.97	9.36
24.	po_task	2274	399,744	3.77	3.22
25.	transmitter	8642	898,768	14.3	7.24
26.	po_task_state	11960	3,052,824	19.8	24.6
27.	entity_state	28569	4,857,328	47.3	39.1
Totals		60,341	12,415,774	100 %	100 %

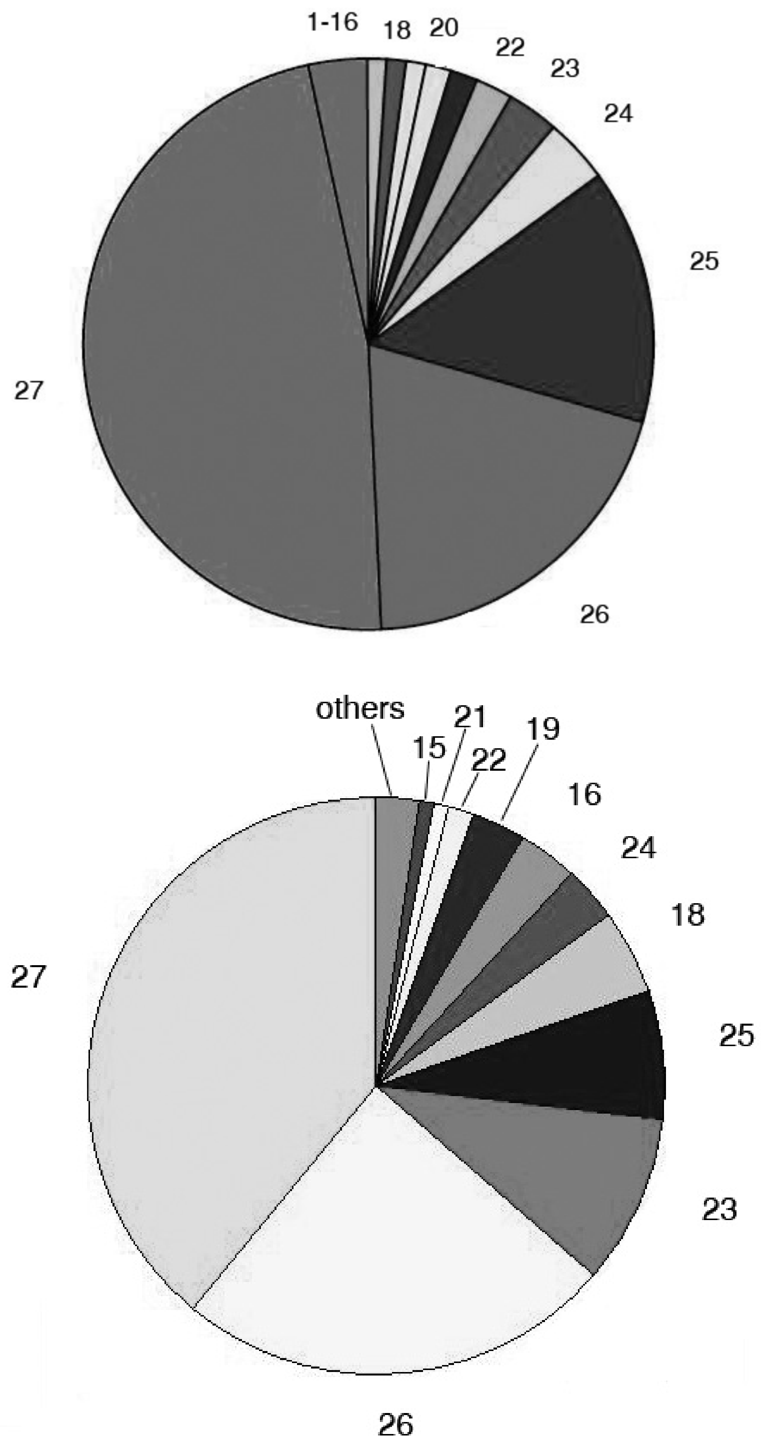


Figure 41: Distribution of Types and Volumes of PDUs Produced in the Simulation of MR1 Vignette. The labels correspond to those of Table 4

It is interesting to note that the percentage of PDU types does not necessarily agree with the percentage of byte volume for the same type. For example, `transmitter` PDUs represent the 14.3 % of the total number of PDUs, but only the 7.24 % of total byte volume, and `po_unit` PDUs are the 2.97 % of type frequency, but the 9.36 % of total byte volume.

The assignment of OTB sites to computer nodes in this simulation is as follows.

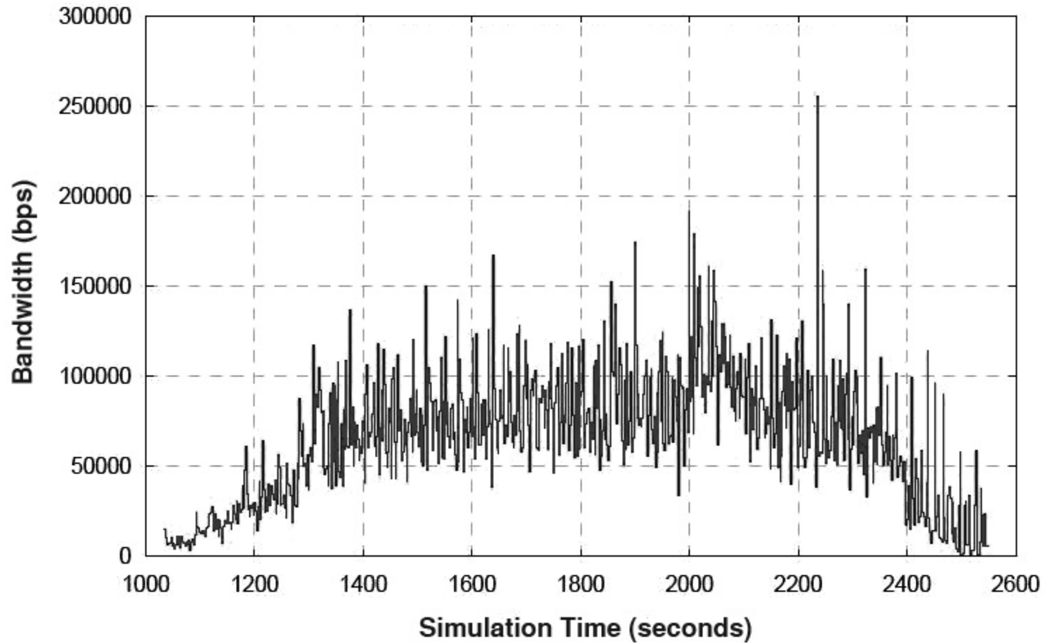
- Site 1519 (0): 50,230 PDUs assigned to plane 0, node 0 (CPU node 0)
- Site 1526 (3): 1,056 PDUs assigned to plane 1, node 0 (CPU node 3)
- Site 1529 (6): 483 PDUs assigned to plane 2, node 0 (CPU node 6)
- Site 1532 (24): 7,382 PDUs assigned to ground station (CPU node 24)
- Site 1533 (9): 553 PDUs assigned to plane 3, node 0 (CPU node 9)
- Site 1538 (12): 637 PDUs assigned to plane 4, node 0 (CPU node 12)

Site #1519 generated 50,230 PDUs 83 %, being it the most preponderant one. The assignment was made such that the site with the highest rate of PDUs belongs to an aircraft and the second largest one in importance goes to the CONUS ground station.

5.6.1.2 Minimum Bandwidth Requirements

Figure 42 shows a more uniform bandwidth requirements than in previous vignettes, but this is mostly caused by the larger number of PDUs in the vignette. As seen, the static analysis indicates that the maximum bandwidth required is near 256 Kbps, but the majority of the time the bandwidth required is less than 200 Kbps.

With an average of near 67 Kbps, it seems that 64 Kbps would be completely insufficient for this vignette. This fact will be acknowledge during the simulation.



Init time = 1034.447	Minimum bandwidth = 773.2
Final time = 2549.808	Maximum bandwidth = 255,745.3
Average = 66,756.4	Std. deviation = 35,082.2

Figure 42: Minimum Bandwidth Requirements in Simulation MR1T6

5.6.2 Slack Time

Figure 43 shows the slack time for all the units (routers and ground station) at 64 Kbps in the wireless channels, and Figure 44 is a zoom in to the Y axis showing that the ground station carries the majority of the negative slack occurrences. This is explained by the fact that the generators onboard the planes are directly connected

to high speed Ethernet buses, while the generator in the ground station is connected to a low speed wireless channel. Figure 44 was plotted by pixels instead of lines to better show the details.

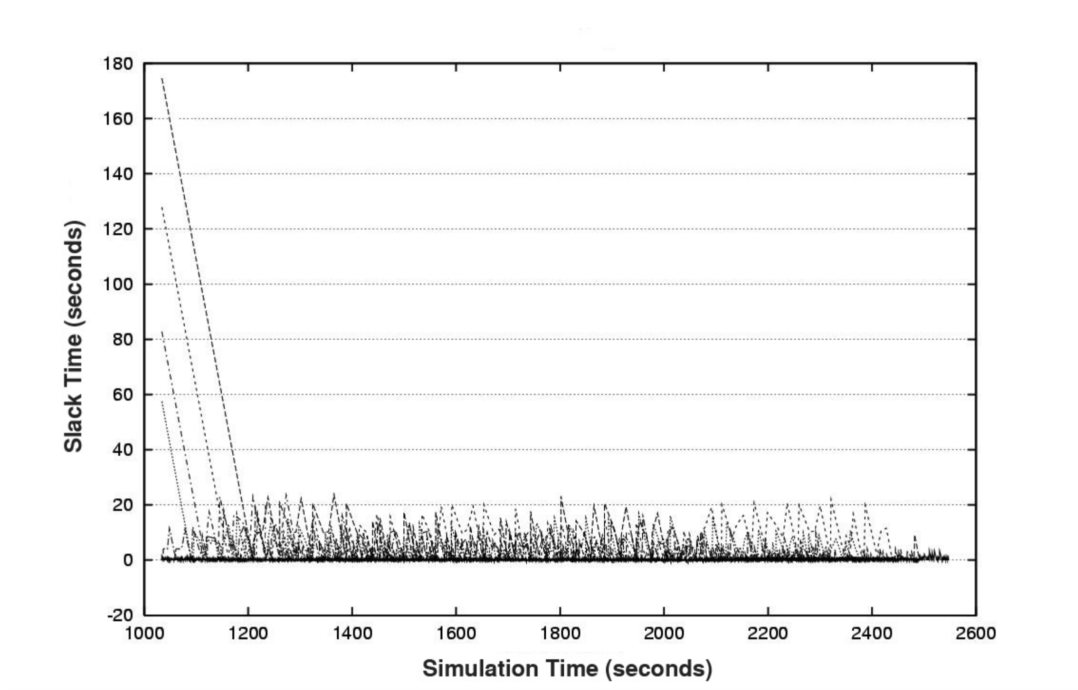


Figure 43: Slack Time to Send Next Message by All the Generators (64 Kbps)

Table 5 displays the percentage of packets with positive slack by site. A separate program was used to calculate them. It calls to the attention the low percentages of positive slack occurrences observed at airplane nodes. At 100 Mbps in the LAN, it is expected to have positive slack occurrences in 95 % or more of the PDUs, but according to the results, the percentage of negative slack occurrences is considerable in airplane nodes. However, those negative slack occurrences are not observed in Figure 43. The reason is that utilizing 100 Mbps in the LAN buses, the negative slack occurrences are almost too negligible to be seen in the graph, but they are still present. The cause of negative spikes is mainly due to the fact that OTB schedules several PDUs at exactly the same time, at least to the resolution of the OTB clock.

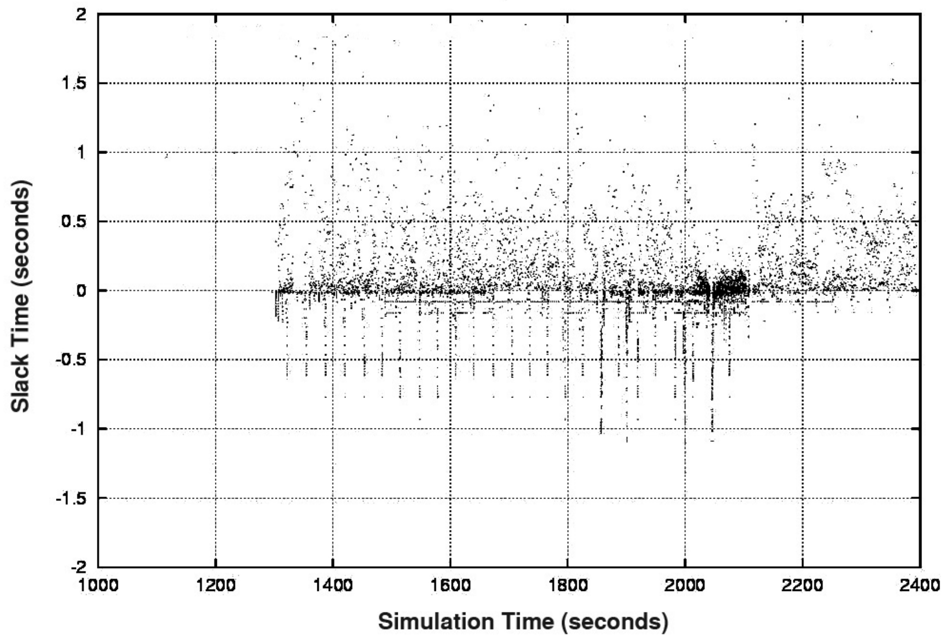


Figure 44: Zoom in of Slack Time Showing Details of Ground Station (64 Kbps)

These negative slack occurrences will not completely disappear by increasing the network bandwidth. One way to reduce or eliminate them without increasing the bandwidth is by bundling and/or rescheduling the PDUs so that they do not occur at the same time.

Figure 45 shows the effect of increasing the wireless bandwidth to 1,024 Kbps in the ground station channel. Although experiments with intermediate values of 128, 200, 256, and 512 Kbps were carried out, it is difficult to visualize them in one Figure. Nevertheless, the consequences of a bandwidth increase are evident: at 1,024 Kbps negative spikes are still present, even though the spike magnitude decreases. The experiments showed that for the other intermediate bandwidths the results are in between, as expected. The more the bandwidth is increased, the less the negative slack is detected.

Table 5: Packets With Positive Slack at Sending Sites in Simulation MR1T6

Site #	Number of PDUs	Percentage	PDUs Sent
0	39,656	78.95%	50,230
3	519	49.15%	1,056
6	236	48.86%	483
9	340	61.48%	553
12	394	61.85%	637
24	4,182	56.65%	7,382
Total:	45,327	75.11%	60,341

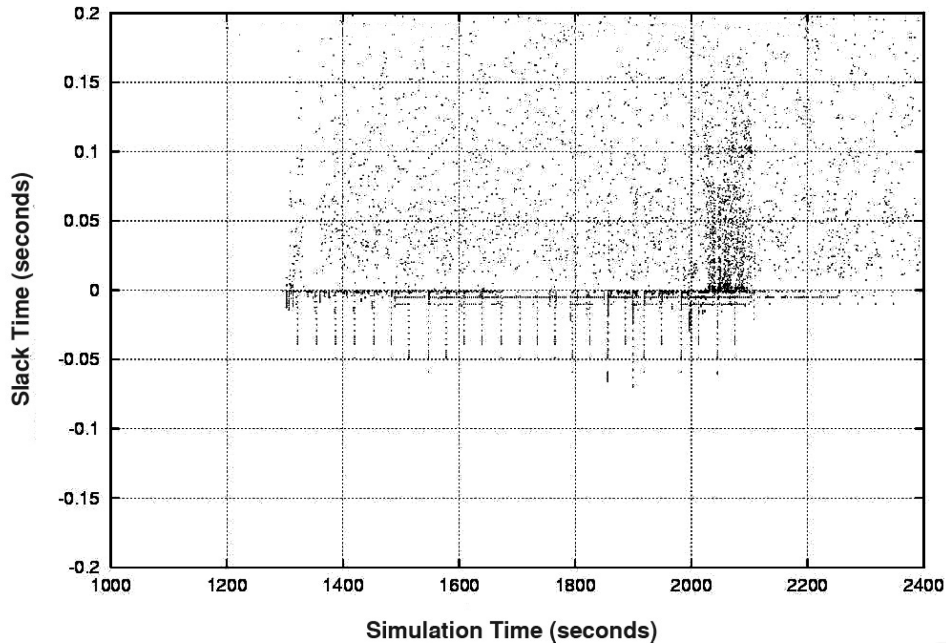


Figure 45: Zoom In of Slack Time to Send Next Message by Ground Station (1,024 Kbps). Negative spikes are still observed.

5.6.3 Travel Time

Figures 46 shows the travel time of PDUs measured by the sink at node 2 in plane 0 using 64 Kbps on the wireless links. At node 2 the graph clearly shows two traces corresponding to two sources of PDUs. Although two traces are visible, only one variable is plotted: the travel time of all PDUs received at node 2.

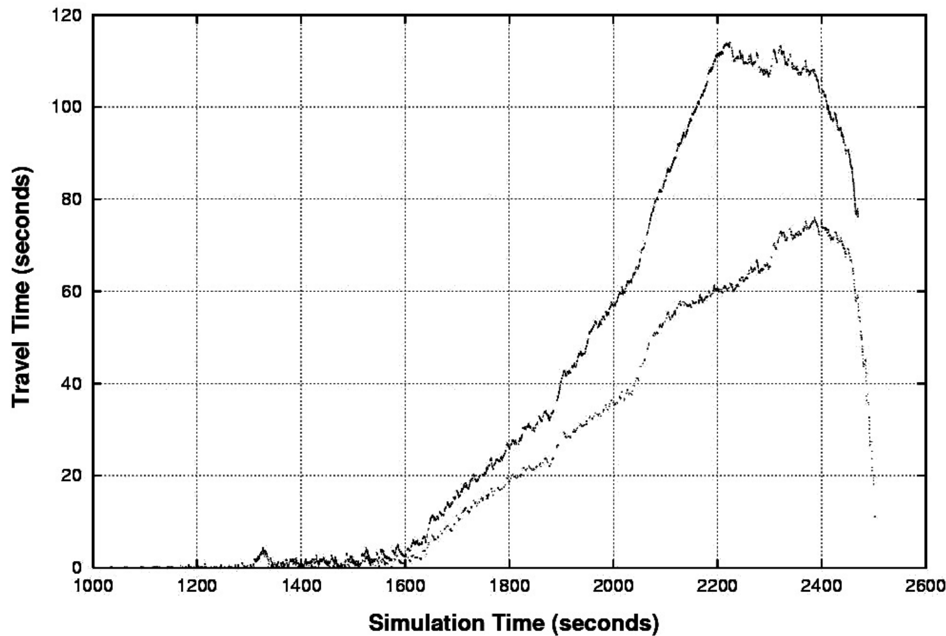


Figure 46: Travel Time at Node 2 in Plane 0 (64 Kbps)

The reason for the two traces is the following. The PDUs that take longer in transit originate at the ground station. These PDUs had to wait on the satellite queue as well as on the router queue. They produce the higher curve. On the other hand, the PDUs coming from computers onboard the other planes had to wait on the router queue only, producing the lower curve. There are no messages coming from nodes within the same plane 0 because of the assignment given. However, if they had been issued, their trace would not be seen because the LAN at 100 Mbps

would render them near zero at the scale used. The graph was drawn using pixels instead of lines to better observe the traces.

Figure 47 shows travel times measured by the ground station using 64 Kbps. Latencies close to 100 seconds are observed during the time interval [2100, 2400] seconds. As a consequence, at 64 Kbps the travel times of most PDUs are completely unacceptable for the OTB simulation requirements. Some PDUs took more than 110 seconds in transit since the time they were sent to the time they arrived.

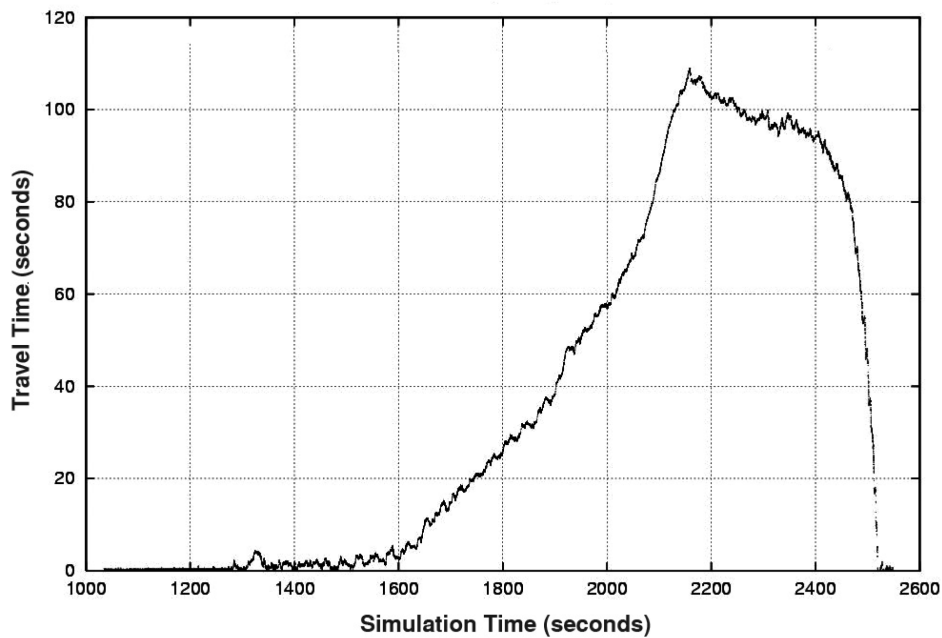


Figure 47: Travel Time at Ground Station (64 Kbps)

Figure 48 shows travel times measured at the ground station for bandwidths of 64 Kbps and 256 Kbps. The graph was zoomed in to the Y axis to show the details in the neighborhood of 0 to 2 seconds. The pixels on the left side (labeled 1) having travel times over 1 second correspond to 64 Kbps, while the other pixels (labeled 2) that almost do not reach the 1 second limit correspond to 256 Kbps. It is worth noting the enormous difference between these two bandwidths. At 256 Kbps in the

wireless channels, the travel times to the ground station are less than 1 second in the majority of cases. Considering that the minimum travel time is about 0.25 seconds, latencies less than 1 second can be acceptable, especially if OTB could deliver the PDUs in a not-so-bursting mode. Latencies less than 1 second will have minimal impact on the fidelity of a distributed simulation.

At 256 Kbps, Figure 48 shows many discrete positive peaks separated at regular intervals that could be diminish by a better scheduling policy. The positive travel time spikes in Figure 48 and the negative slack time spikes shown in Figure 44 and Figure 45 are correlated. This is because the more time the sending site is behind the timestamped schedule, the more heavy the network traffic is and the packets will have to wait more time in router queues. Both measures are good indicators of the network performance. If some PDUs at the spikes of negative slack could be moved to time intervals of positive slack, the network traffic would become less bursty. We will investigate this further in later sections.

5.6.4 Queue Length

The two most important queues to analyze are the queue at the router onboard plane 0 and the queue at the satellite, because these routers are the most heavily loaded in the simulation. The router at plane 0 connects a high speed link of 100 Mbps to a slow link ranging from 64 Kbps to 256 Kbps. Therefore, messages coming from plane 0 will wait at the router queue for a chance to be transmitted. The satellite receives and transmits all the messages at low speeds, which constitutes a bottleneck in the system. The routers at other planes not heavily transmitting PDUs take packets from a slow wireless channel and pass them to a fast Ethernet link, resulting in almost no queue waiting time. Nevertheless, the queue at another router

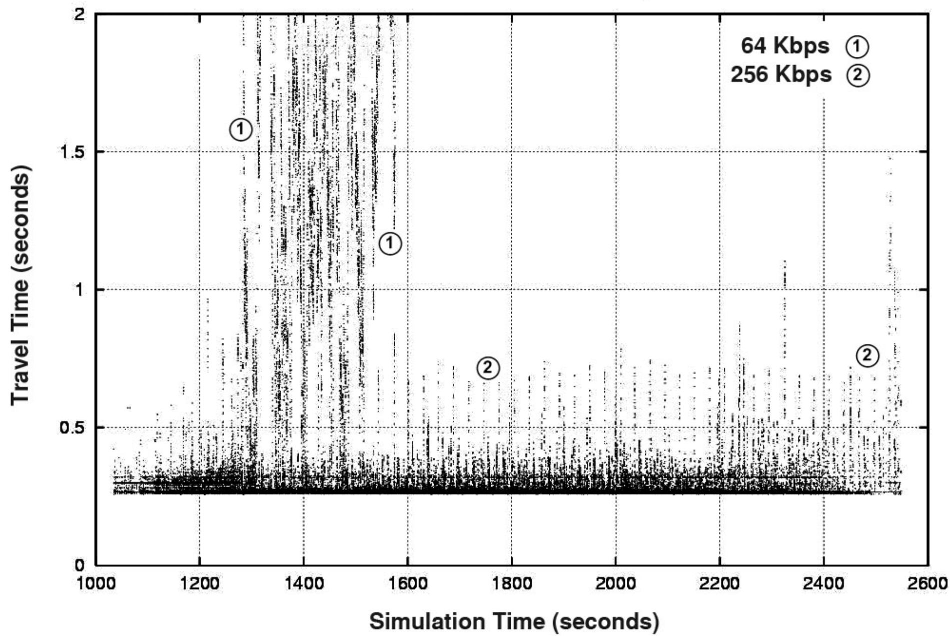


Figure 48: Zoom In of Travel Times at Ground Station (64, 256 Kps)

is shown as a sample of the behavior of the other routers. Figure 49 represents the number of messages in the router at node 0, using 64 Kbps in the wireless channels. The queue length becomes really unacceptable, reaching more than 3,000 messages during some periods.

On the contrary, Figure 50 shows that the router at plane 3 has a typical queue with a maximum of 23 messages. The reason for the short queue is that the corresponding node 9 transmits only 553 PDUs, which are easily handled by the router. Figure 51 shows that the queue at the satellite at 64 Kbps has an unacceptable behavior with a peak of more than 2,200 messages. Figure 52 shows the effect on the queue length of the router onboard plane 0 when the bandwidth increases from 64 Kbps to 256 Kbps. As a result, the queue length decreases to less than 50 messages in its highest peak. Therefore, the change of link speed greatly reduces the router queue.

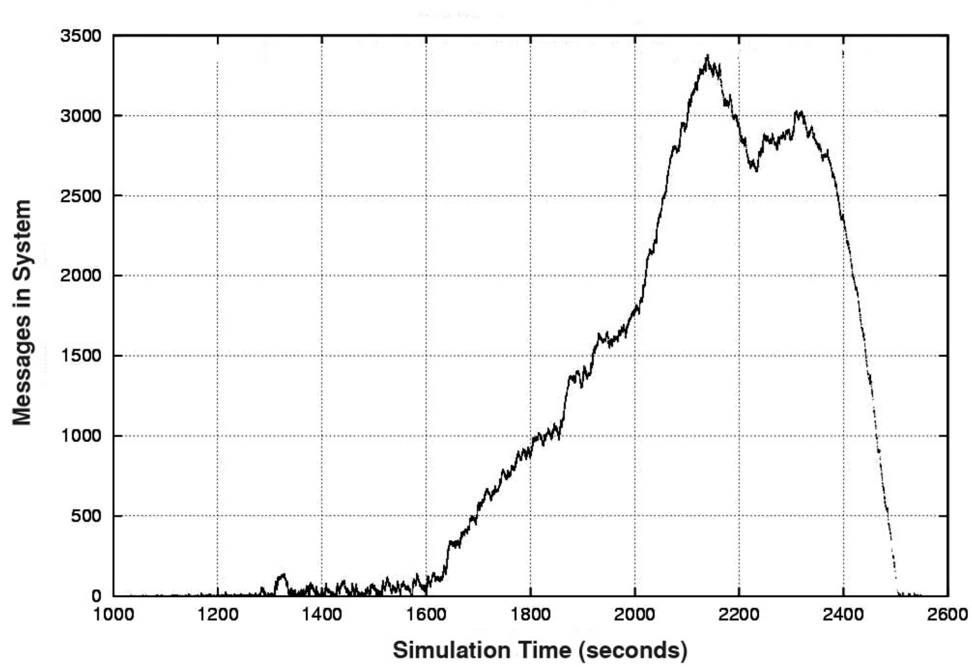


Figure 49: Messages in System at Plane 0 (64 Kbps)

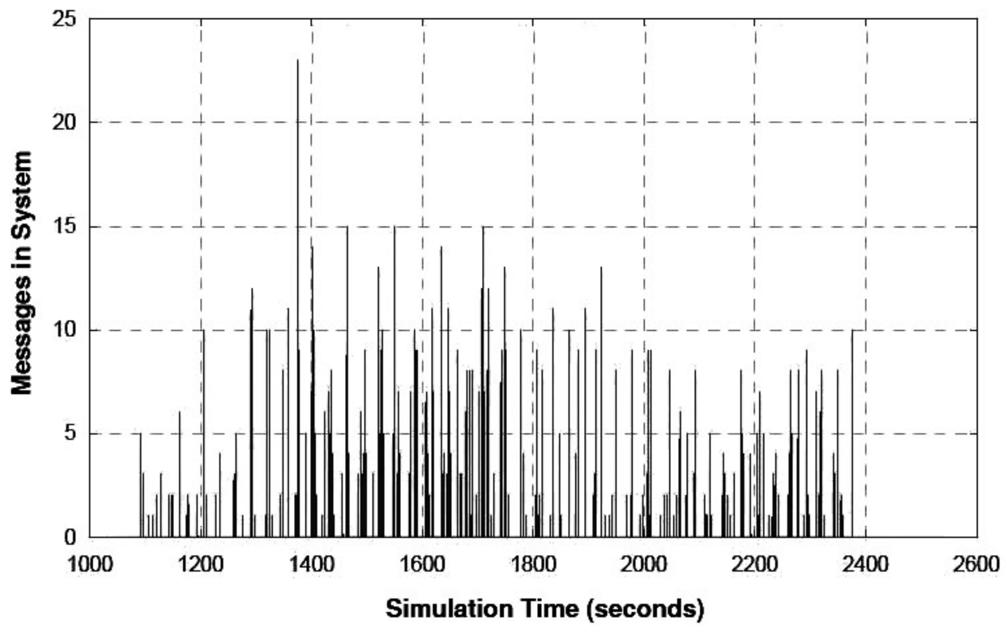


Figure 50: Messages in System at Plane 3 (64 Kbps)

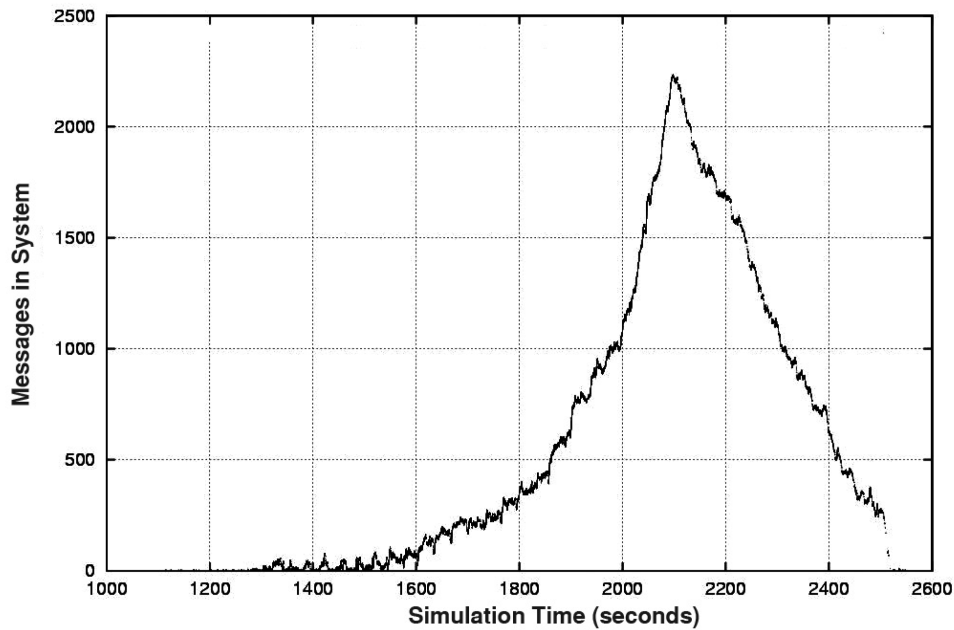


Figure 51: Messages in System at Satellite (64 Kbps)

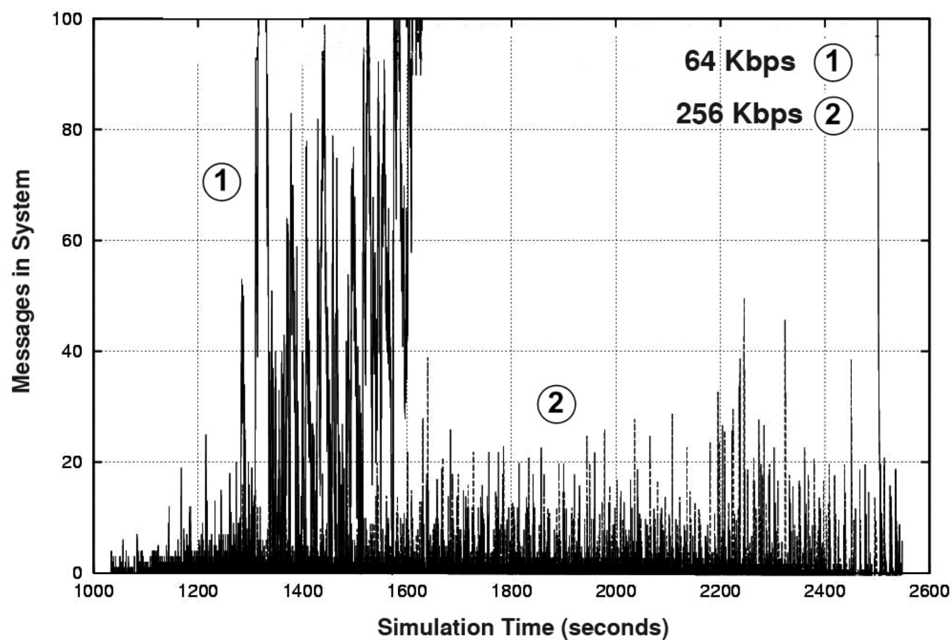


Figure 52: Zoom In of Messages in System at Plane 0 (64 Kbps and 256 Kbps)

At the satellite the change in the queue length is also noticeable, as shown in Figure 53. Fewer than 25 messages are held in the satellite during the highest peak for a bandwidth of 256 Kbps.

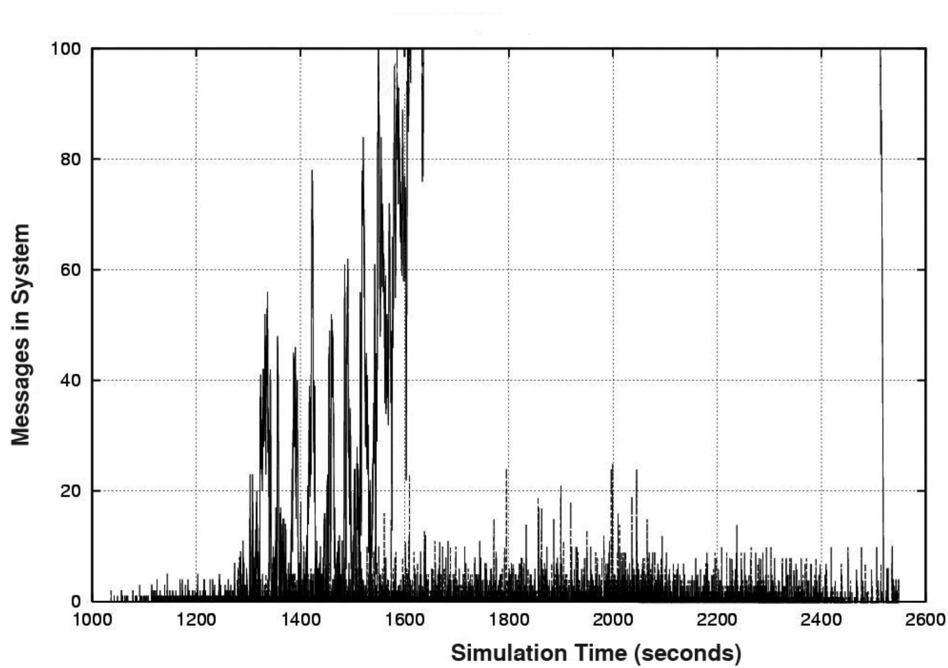


Figure 53: Zoom in of Messages in System at Satellite (64 Kbps and 256 Kbps)

5.6.5 Collisions

There are two factors that influence the way collisions are detected and analyzed in the OMNeT simulator. First, the simulator considers the transmission media as an ideal bus. In other words, it is completely possible for a collision to be detected at some point of the bus and not at others. Due to the high propagation speed of the bus, for short distances between most nodes, like in the LAN bus or the WPP bus, this behavior is irrelevant. However, for the WSP channel, it has to be taken into account. Second, the bus was programmed such that a message delivered at some

bus gate is not returned back to the sending entity, even if it collided. Therefore, in order to detect collisions, a plain listener node (sentinel) should be chosen, as the router at plane 7 in this simulation.

Figure 54 shows the collision accumulation sensed at planes 1, 2 and 7. Planes 1 and 2 are transmitters and receivers, while plane 7 is a receiver only. A few more than 6,000 collisions were detected at 64 Kbps in plane 7, which represents approximately 10 % of the total number of PDUs. The statistics are very similar in the three cases, but plane 7 detects more collisions because the other planes cannot detect collisions caused by the sending of their own messages.

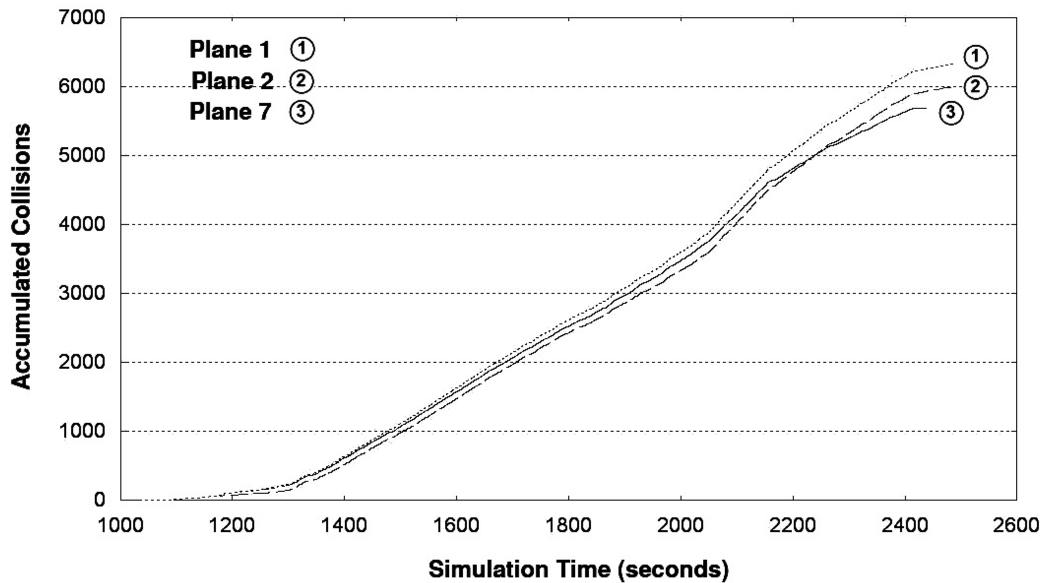


Figure 54: Collision Accumulation at Planes 1, 2 and 7 (64 Kbps)

Figure 55 shows the collisions per second in the WSP wireless channel. The other two links are not displayed because the LAN bus has no observable collisions and the WPP link exhibits just a few ones. The WSP channel gets most of the collisions. At 64 Kbps, the highest rates are close to 13 collisions per second in this

channel, near the second 2,100, with an average of approximately 4 collisions per second for the whole simulation.

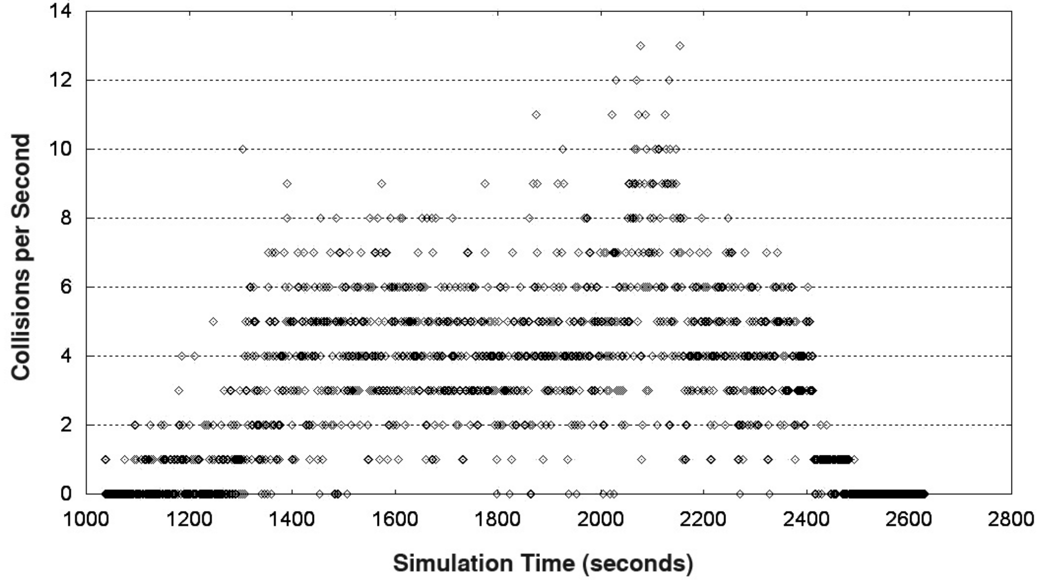


Figure 55: Collisions per Second Detected at the WSP Wireless Channel in Plane 7 (64 Kbps)

Simulations performed using combinations of 64, 200, 256, 512 and 1,024 Kbps in wireless channels showed that the number of collisions decrease when the bandwidth increases, as expected. Collision accumulation statistics viewed from plane 7 are given in Figure 56. At 256 Kbps the total number of collisions is near 3,800 that represents about 6 % of all the PDUs.

5.6.6 Spike Analysis of Slack Time

Figure 57 shows a zoom in sample of the slack time at the ground station when the model is executed at 64 Kbps for wireless channels. Some negative spikes

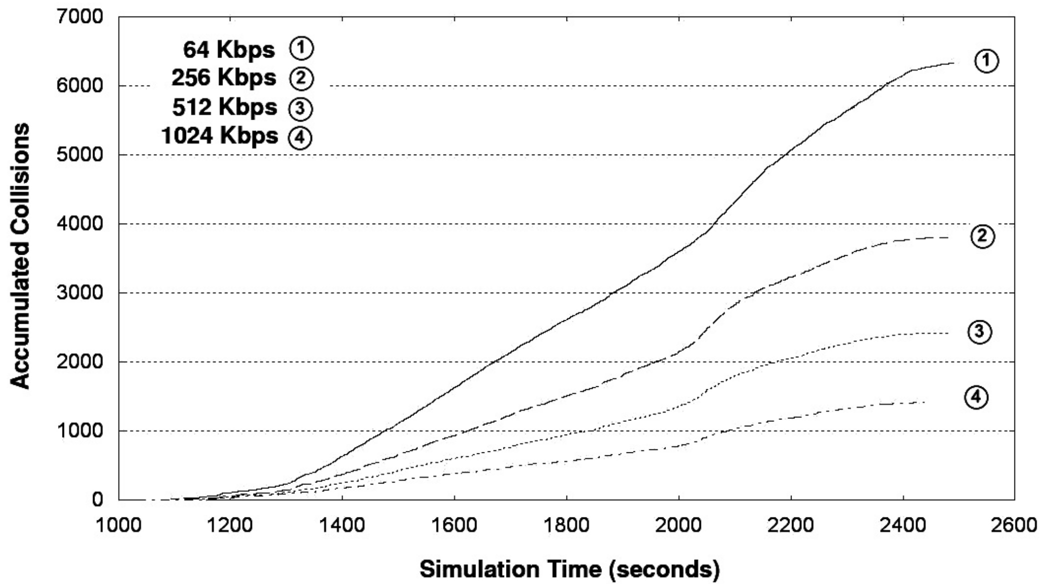


Figure 56: Collision Accumulation at Plane 7 During Simulation MR1T6 (64, 256, 512, 1,024 Kbps)

are visible at regular time intervals. Positive points indicate that the bandwidth is sufficient to handle the PDUs in the neighborhood previous to the point. On the contrary, negative spikes indicate a lack of bandwidth in the wireless channels. Those spikes deserve more attention to understand and design a new protocol to correct the problem. The OTB simulator produces the negative spikes when multiple PDUs are scheduled at the same or almost the same time. Identification of the PDUs responsible for the negative spikes is the first step towards the study and possible modification of the OTB scheduling policy in the next Chapter.

Because the phenomenon seems to be cyclic, one initial approach to explain it relays on the analysis of the different PDUs participating in the spike, correlating them with actions occurring in the vignette at those times. The sample includes the spikes captured in the time interval [1400, 1600] seconds. This is a representative sample of spikes produced at the generators of PDUs. The spikes were studied at

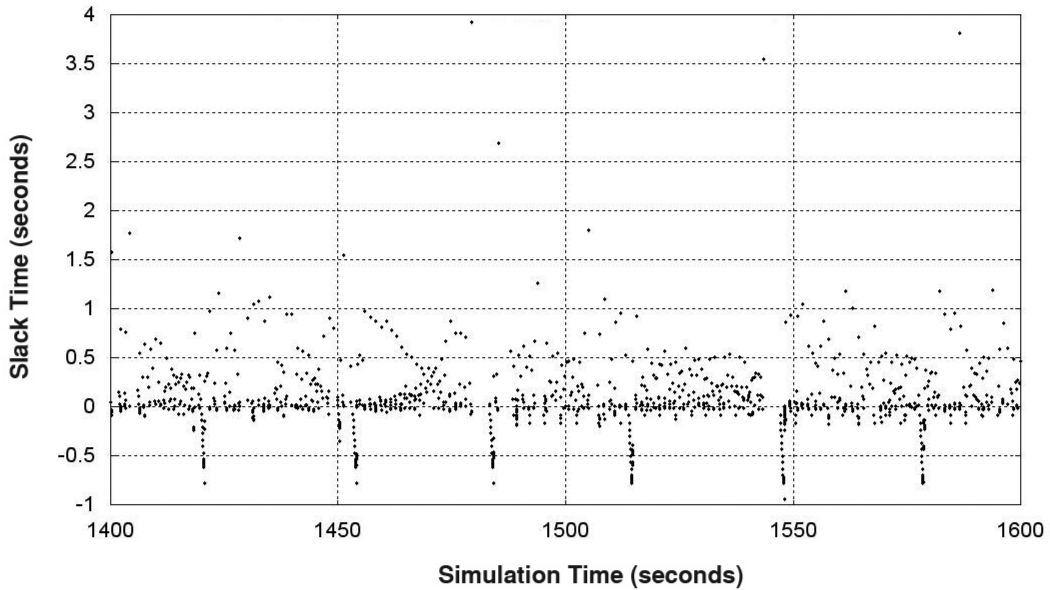
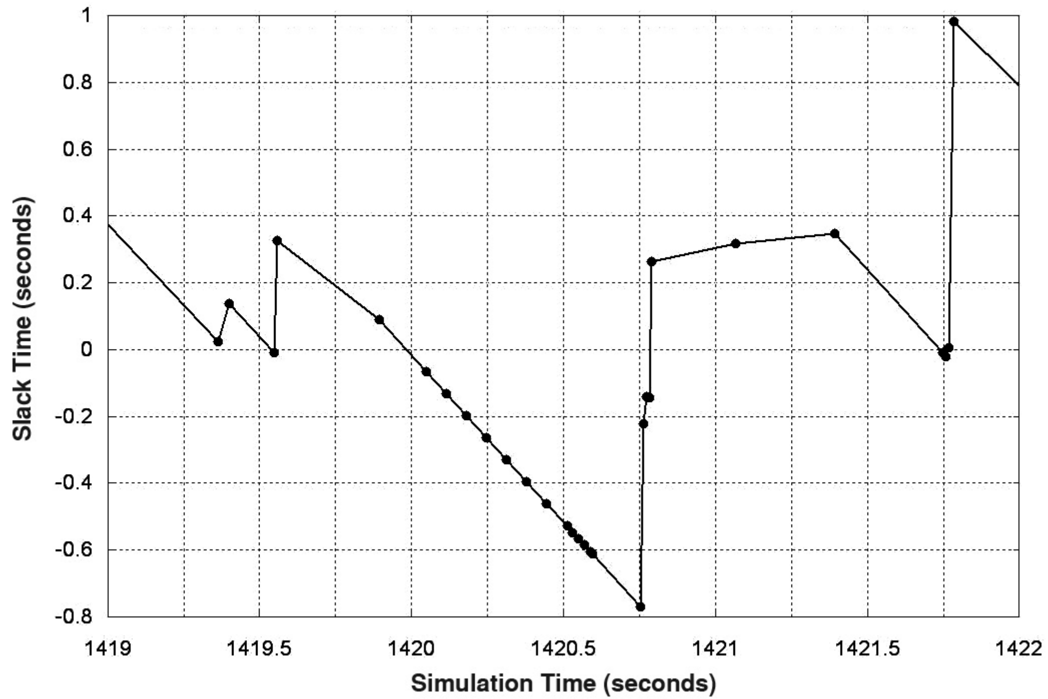


Figure 57: Slack Time at Ground Station Showing Negative Spikes (64 Kbps)

64 Kbps. Higher rates cause a decrease in the magnitude of the negative spikes, but the spikes are still present because they are caused mainly by OTB scheduling policies. A total of six relevant negative spikes were studied and two of them are presented here in the next paragraphs.

5.6.6.1 Spike at second 1420

Figure 58 shows the participating PDUs responsible for the negative spike, along with a close up of the spike graph. It is worth observing that eight `po_fire_parameters` PDUs were issued at the same time, as well as four `po_line` PDUs, among others.



Size	Timestamp	PDU Type	Size	Timestamp	PDU Type
84	:23:39.536	point	152	:23:39.982	line
84	:23:39.536	point	152	:23:39.982	line
80	:23:39.883	task_state	152	:23:39.982	line
528	:23:39.982	fire_parameters	152	:23:39.982	line
528	:23:39.982	fire_parameters	56	:23:39.982	task_state
528	:23:39.982	fire_parameters	1,272	:23:39.982	task
528	:23:39.982	fire_parameters	80	:23:39.982	task_state
528	:23:39.982	fire_parameters	80	:23:40.540	task_state
528	:23:39.982	fire_parameters	80	:23:40.633	task_state
528	:23:39.982	fire_parameters	56	:23:40.639	task_state
528	:23:39.982	fire_parameters			

Figure 58: Negative Spike at Second 1420 Showing Participating PDUs

5.6.6.2 Spike at second 1454

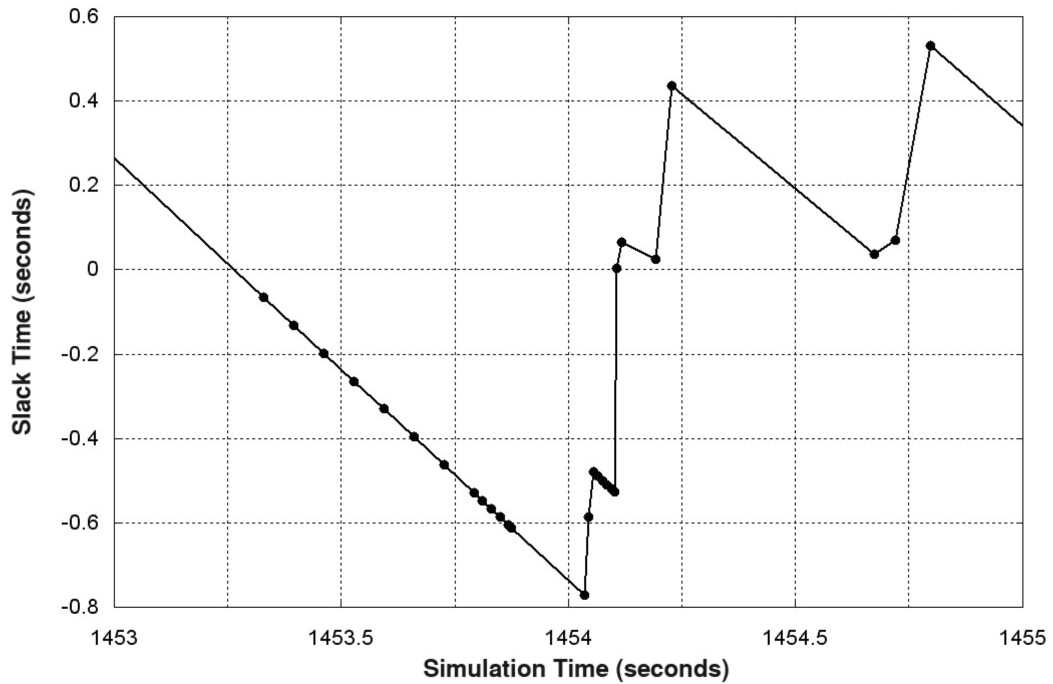
As with the previous spike, Figure 59 shows that at second 1454 eight `po_fire_parameters`, four `po_line`, five `po_task`, and five `po_task_state` are responsible for this spike. The spikes at seconds 1484, 1514, 1548 and 1578 are very similar to the previous ones, and will not be elaborated on individually.

5.6.7 Conclusions of Simulation MR1T6

As predicted by the independent analysis, 64 Kbps in the wireless links is completely insufficient to handle traffic in the interval [1600, 2550] seconds. Latencies of more than 70 seconds were detected at plane 0 for traffic coming from other planes, and more than 110 seconds if the traffic originated at the ground station. As predicted, a big improvement was achieved starting at 200 Kbps. Latencies less than 1 second were almost always the rule for messages received at the ground station.

The increase in bandwidth from 64 Kbps to 256 Kbps had a major impact in the router and satellite queue lengths. At the router in plane 0, the queue length changed from 3,400 messages to fewer than 50 messages at the highest peak, for a 98 % reduction. At the satellite, the queue lengths changed from 2,200 messages to fewer than 25 messages at the highest peak, representing a reduction of 99 %.

Relatively few collisions were detected, which are not enough to significantly change the results or conclusions. A summary of the total number of collisions is given in Table 6. As clearly seen in the table, collisions decrease as the bandwidth increases, which is explained in virtue of the lesser time required to transmit each packet and the corresponding lesser probability of collision.



Size	Timestamp	PDU Type	Size	Timestamp	PDU Type
528	:24:13.263	fire_parameters	56	:24:13.263	task_state
528	:24:13.263	fire_parameters	1,272	:24:13.263	task
528	:24:13.263	fire_parameters	80	:24:13.263	task_state
528	:24:13.263	fire_parameters	80	:24:13.458	task_state
528	:24:13.263	fire_parameters	80	:24:13.574	task
528	:24:13.263	fire_parameters	80	:24:13.574	task
528	:24:13.263	fire_parameters	80	:24:13.574	task
528	:24:13.263	fire_parameters	80	:24:13.574	task
152	:24:13.263	line	48	:24:13.574	task_state
152	:24:13.263	line	48	:24:13.574	task_state
152	:24:13.263	line	56	:24:14.109	task_state
152	:24:13.263	line			

Figure 59: Negative Spike at Second 1454 Showing Participating PDUs

Table 6: Total, Relative Percentage and Average Number of Collisions per Second in Simulation MR1T6

Bandwidth Kbps	Collisions	Percentage	Frequency coll/sec
64	6,320	10.5 %	4.2
200	4,434	7.3 %	2.9
256	3,804	6.3 %	2.5
512	2,421	4.0 %	1.6
1,024	1,416	2.3 %	0.9

The negative spikes in slack time studied in Simulation MR1T6 are very similar. All of them include three main types of PDUs: `po_fire_parameters`, `po_line`, and `po_task_state` PDUs. In all cases, sequences of these PDUs were scheduled exactly at the same time, causing the spike. It seems that the main sequence of PDUs in a negative spike is of type `po_fire_parameters`, because an entity at the ground station is firing against some enemy at regular time intervals.

5.7 Simulation MR1GS: Vignette MR1 Revisited

Given that site #1519 assigned to node 0 in plane 0 for Simulation MR1T6 generates 83 % of all the PDUs, it seemed interesting to assign it to the CONUS ground station connected to the satellite via a wireless channel. The idea was suggested by PEO STRI personnel during an update presentation. The results obtained were certainly interesting, and leaded to the developing of the bundling algorithm described in Section 4.4.

5.7.1 Independent Analysis and PDU Assignment

This simulation makes use of the same data set as in Simulation MR1T6, and so the types of PDUs, data volumes and percentages remain the same. The only difference is that the assignment of sites #1519 and #1532 corresponding to computer 0 and ground station was swapped. Therefore, the assignment of sites to computers in this experiment is as follows.

Site 1532 (0):	7,382 PDUs	assigned to plane 0, node 0
Site 1526 (3):	1,056 PDUs	assigned to plane 1, node 0
Site 1529 (6):	483 PDUs	assigned to plane 2, node 0
Site 1533 (9):	553 PDUs	assigned to plane 3, node 0
Site 1538 (12):	637 PDUs	assigned to plane 4, node 0
Site 1519 (24):	50,230 PDUs	assigned to ground station

The minimum bandwidth requirements are the same as in Simulation MR1T6 and can be found in Figure 42.

5.7.2 Slack Time

As seen in Figure 60, the most noticeable feature is the enormous negative slack (-75 seconds) in the ground station at 64 Kbps. These results are as expected because the ground station has to transmit a large number of PDUs across a relatively slow bus.

Figure 61 shows the slack time at the ground station when the bandwidth is set to 128 Kbps. Just by increasing the speed from 64 Kbps to 128 Kbps, the negative slack time changes dramatically from values close to -75 seconds to values near -1.5

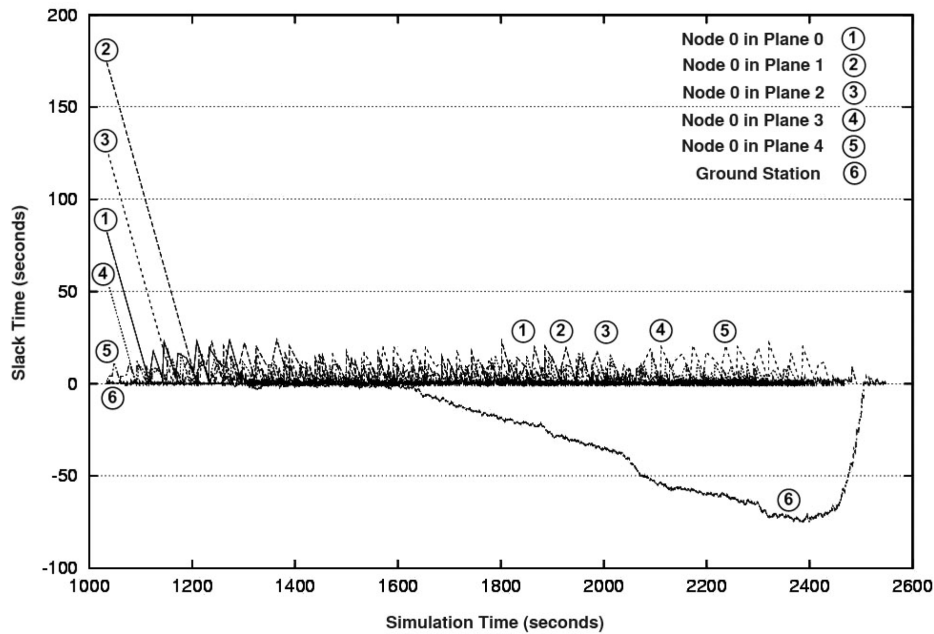


Figure 60: Slack Time to Send Next Message at Planes 0, 1, 2, 3, 4 and Ground Station (64 Kbps)

seconds. It is worth noting that a regular pattern of negative spikes is observed at intervals of approximately 28 seconds.

Figure 62 is a zoom in to the Y axis of the slack time at the ground station. The graph clearly show that most of the time the negative slack falls into the interval [-0.4, 0]. Numerous negative spikes of all sizes can be seen in an apparently regular distribution during the majority of the simulation time.

The positive slack time has a different behavior. As seen, positive spikes are not produced. The reason is that when a positive slack is detected, the simulator waits that time before processing the PDU, and the following PDUs are not read yet. When the simulator is ready to process the next PDU, some time has elapsed. Even if the second PDU has a positive slack, in the graph the points corresponding to the two consecutive PDUs are separated at least by the waiting time, and so a column or spike is not created.

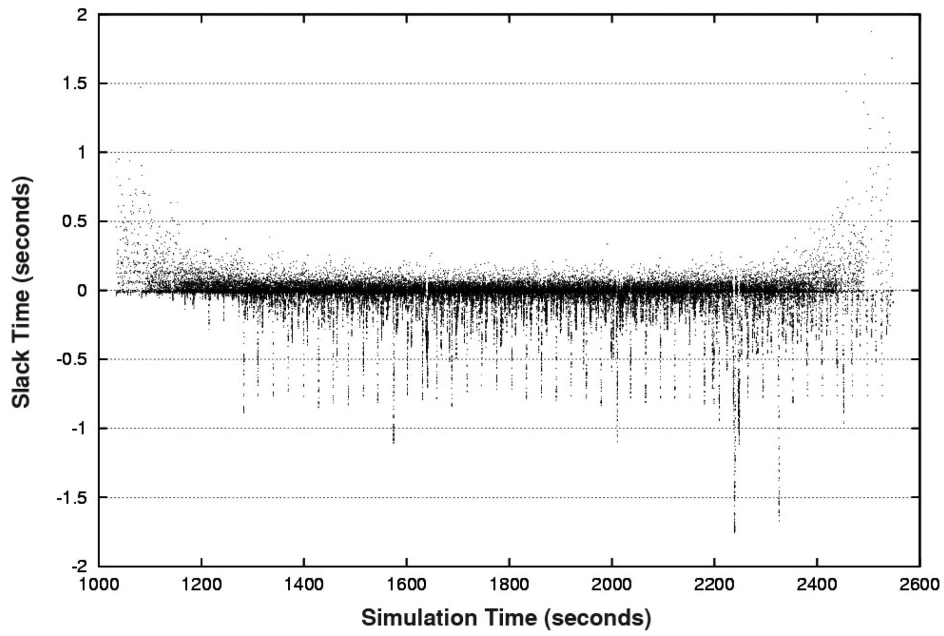


Figure 61: Slack Time to Send Next Message at Ground Station (128 Kbps)

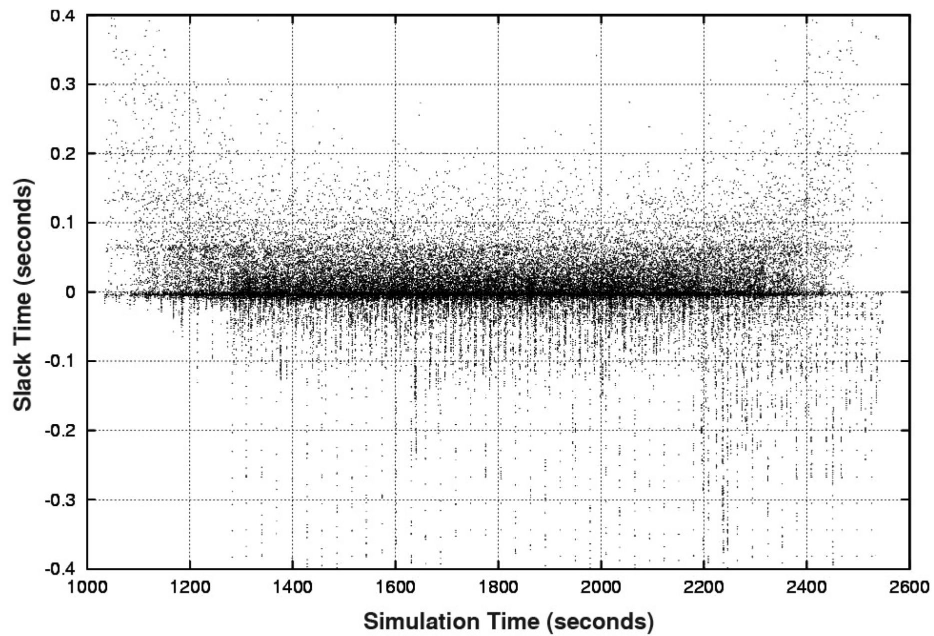


Figure 62: Zoom In of Slack Time to Send Next Message at Ground Station (256 Kbps)

5.7.3 Travel Time

Figure 63 shows the travel time of PDUs measured at node 0 in plane 7, using 64 Kbps in wireless links. The increasing curve results from PDUs originated at the ground station that waited unbounded latencies at the satellite queue. On the other hand, the PDUs coming from nearby airplanes arrived with a negligible time delay for the scale used.

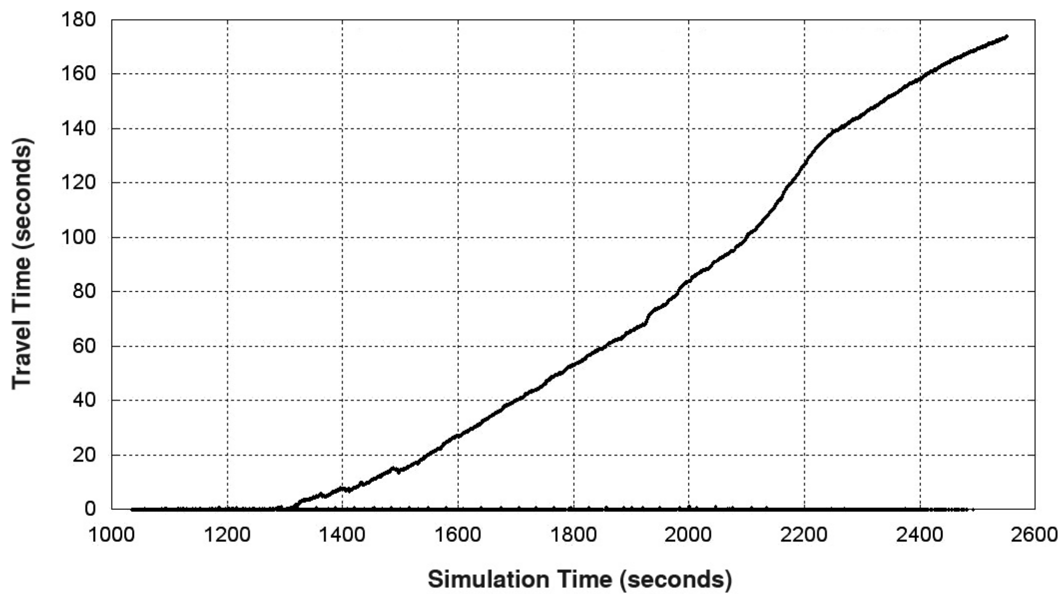


Figure 63: Travel Time at Plane 7 (64 Kbps)

Increasing the wireless bandwidth to 200 Kbps or more produces an enormous change in the travel time. Figure 64 displays travel times at 256 Kbps. All of the plotted PDUs fall below the level of 0.5 seconds. The graph indicates that the PDUs belong to two different subsets. The first subset corresponds to PDUs sent by the ground station. These PDUs needed 0.255 seconds to travel the earth-satellite-earth distance plus their waiting time in satellite and router queues. The second subset is

made up of PDUs coming from other airplanes. These PDUs waited in the router queues only.

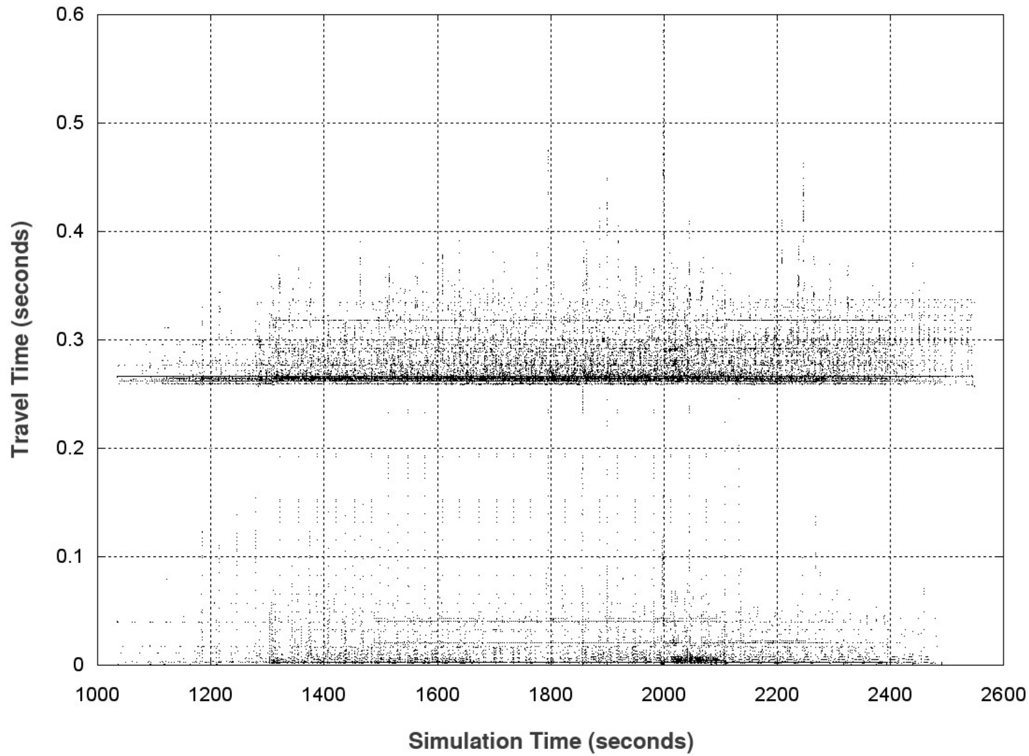


Figure 64: Travel Time at Plane 7 (256 Kbps)

From the graphs, it can be concluded that the change in bandwidth from 64 Kbps to 256 Kbps is a key factor in the overall network performance, a conclusion that was already predicted by the offline independent analysis.

5.7.4 Queue Length

Figure 65 plots the queue length at the router in plane 0 for a wireless bandwidth of 64 Kbps. The queue length is manageable, even at 64 Kbps, where the maximum

number of messages in the system is less than 80. At higher speeds the queue become noticeable shorter.

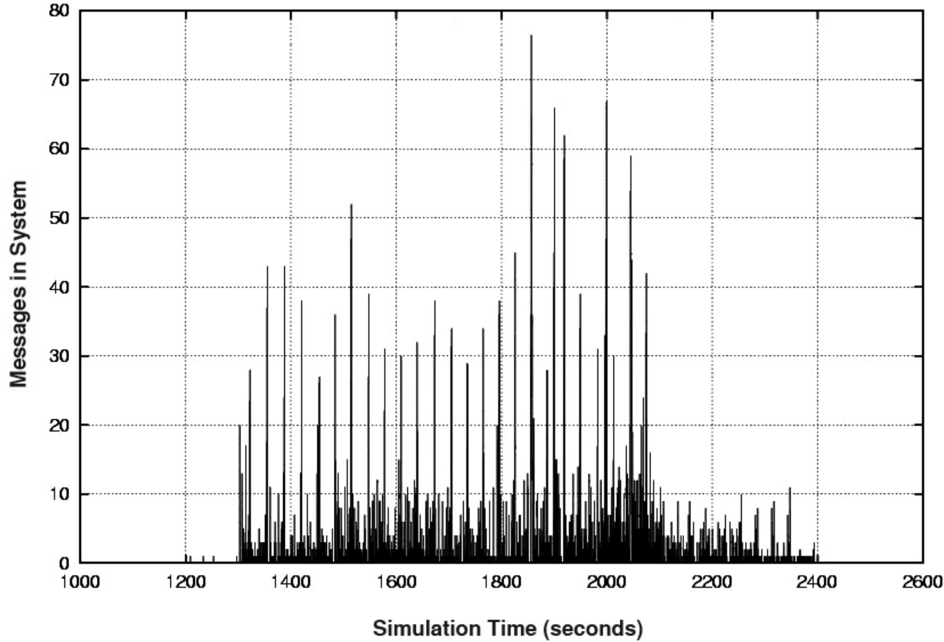
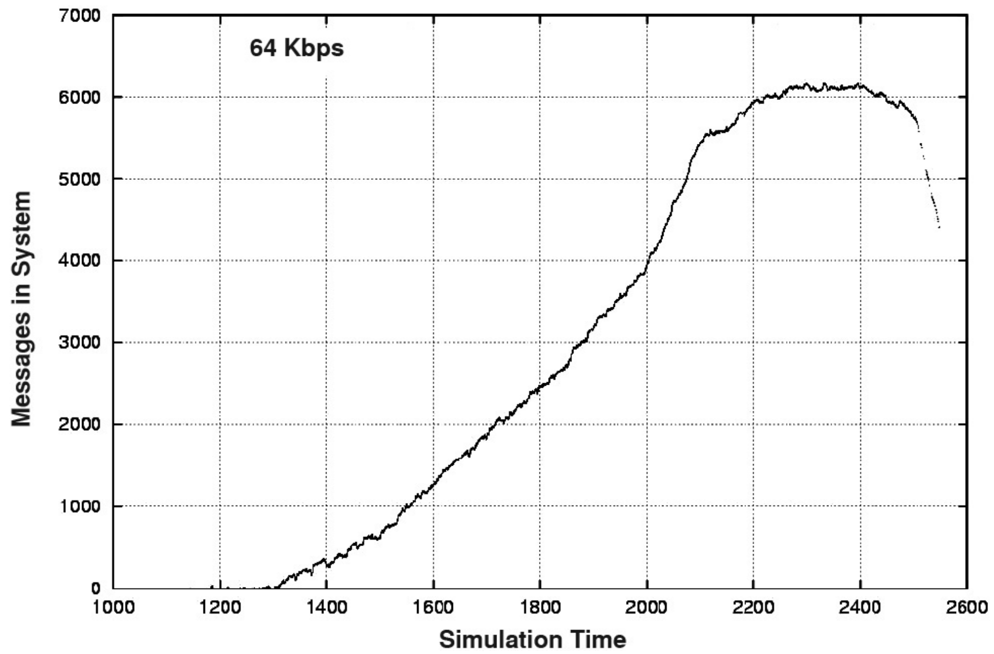


Figure 65: Messages in System at Plane 0 (64 Kbps)

However, Figure 66 shows that at 64 Kbps the satellite queue becomes extremely long, reaching values over 6,000 messages. Also, it is seen that the improvement from 64 Kbps to 256 Kbps is significant, requiring storage for 35 messages only at the highest peak. This length is well-handled and achievable, especially if the bundling and replication algorithms proposed in Section 4.4 are implemented.

By setting the wireless channels to a bandwidth of 1,024 Kbps, Figure 67 exhibits what can be considered as an upper bound on the performance achievable on the satellite queue. As observed, a satellite queue of fewer than 20 messages is very difficult to achieve for the MR1 vignette with the current technology and algorithms.



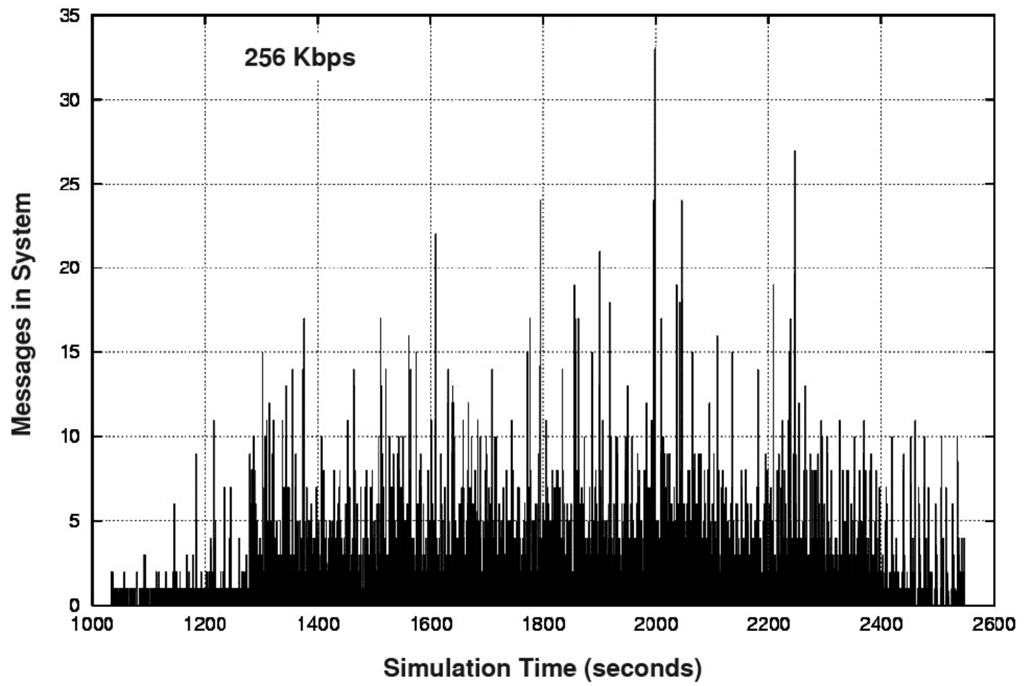


Figure 66: Messages in System at the Satellite (64 Kbps and 256 Kbps)

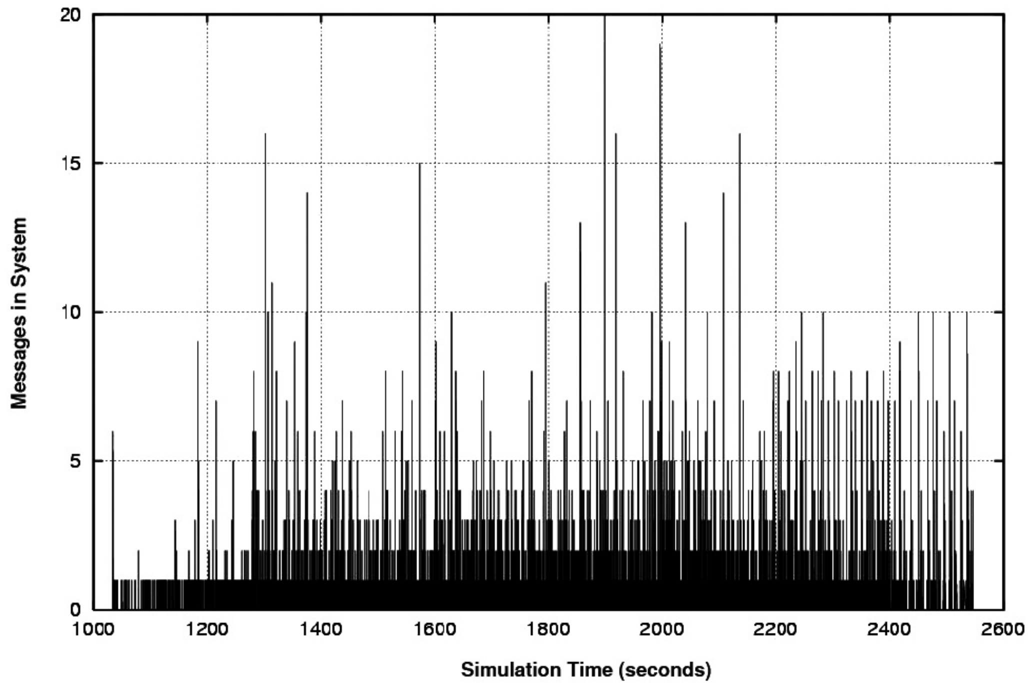


Figure 67: Messages in System at the Satellite (1,024 Kbps)

5.7.5 Collisions

Figure 68 is the summary of collision accumulation as seen by the sentinel node on plane 7 at 64, 256, 512 and 1,024 Kbps. At 64 Kbps the number of collisions represents approximately 8 % of the total number of PDUs, while at 256 Kbps the percentage descends to 5 %. As with Simulation MR1T6, collisions are small enough not to cause an important change in the rest of the statistics without implementing an exponential backoff treatment.

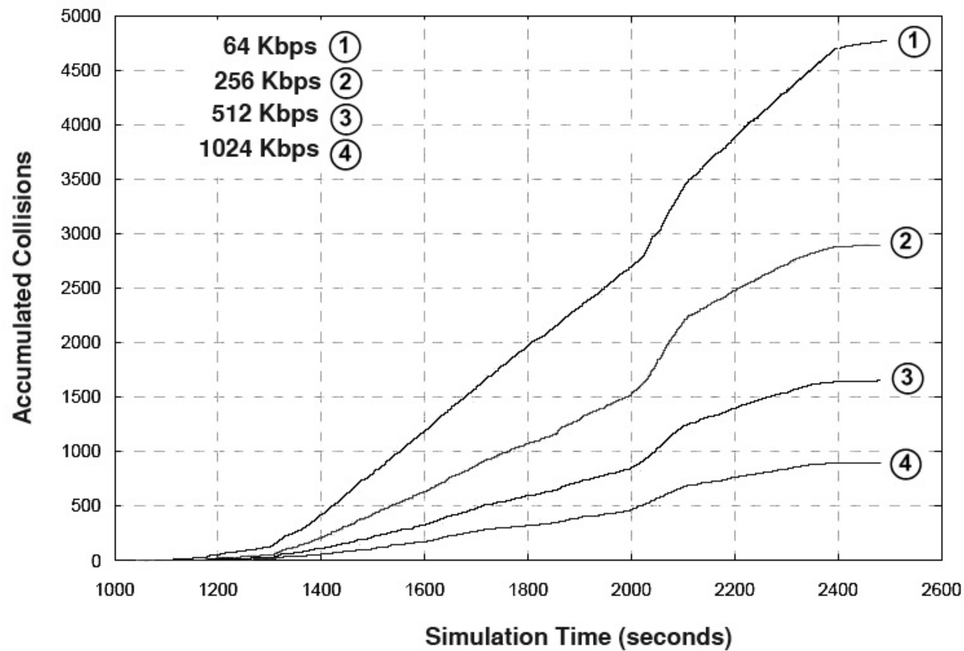


Figure 68: Collision Accumulation at Plane 7 (64, 256, 512, 1,024 Kbps)

5.7.6 Conclusions of Simulation MR1GS

The assignment of site #1532 to the ground station moves the most active computer to this CONUS station, making the bandwidth of the WGS link a key point for network performance assessment. At 64 Kbps, extremely negative time slack occurrences are produced. The reason is that the transmission time of the generator in the ground station is limited by the low bandwidth and cannot schedule the PDUs as indicated by their timestamps. In other words, the low bandwidth restriction acts as a contention mechanism. Increasing the wireless bandwidth to 256 Kbps produces an enormous change in the general performance of the system, result that was predicted by the independent analysis.

5.8 Simulation using Head-of-Line Strategy

If a queue includes the Head-of-Line (HoL) discipline, then its elements are assigned a priority. All the elements of the same priority constitute a class and they are serviced under a FIFO policy within the class. Different classes belong to different queues, and the system is then modeled as a multi-dimensional queue. The system is equivalent to a single priority queue in which those elements of higher priorities are moved to the front of those elements with less priorities upon arrival.

HoL strategies are mentioned in Liu's Ph.D. dissertation [Liu02] where an ATM switch with input queue can get blocked due to a HoL cell waiting for an occupied output port, in the Dumas' paper [DGR01] where the concept of effective bandwidth for a queue with two priority levels is investigated in relation with the admission control for ATM switches, in Guillemin [GM04] where a multi-dimensional preemptive resume queue with HoL priority service is studied, and in the research papers [PW03] and [LS93], among many others.

In order to study the effect of HoL, the OMNeT simulator was modified to include HoL priority service in all router and satellite queues. The queues were declared as sorted queues, and so OMNeT maintains them using a stable sorting policy. Sorting stability ensures that within the same priority class, the PDUs are serviced in a FIFO order. The sort order is based on the PDU type. After analyzing the PDUs participating in negative slack spikes, the assignment of priorities was established as described in Table 7.

The priorities range from 0 to 9, with 9 being the highest priority. PDUs having priority 9 were systematically found forming sequences during negative spikes and, therefore, the sense of urgency is derived from this fact. The effects of an HoL service need to be more succinctly measured by the OMNeT simulator since it does not interpret the packets and react to them. The simulator task is to deliver all

Table 7: PDU Priorities for HoL Service

PDU type	Priority
po_fire_parameters	9
po_line	9
po_task	9
po_task_state	7
entity_state	1
all others	5

the packets bounded by the bandwidth in each link. Once the packets arrive to their destinations, the simulator loses track of them. If due to a higher priority, some packets are delivered first, the simulator records statistics about their latency, queue lengths, collisions, etc., but not about the effect of this change of predefined order at the destinations.

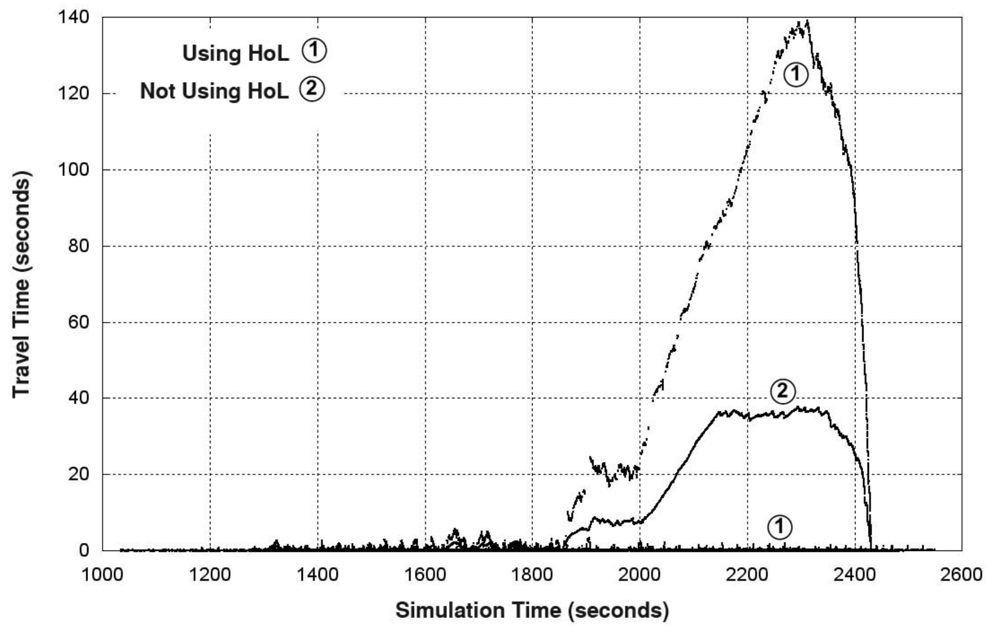
Most statistics collected during the HoL simulation were very similar to those without using HoL. For example, the negative spikes observed in Figures 57 and 62 are the same under the HoL discipline. The reason is that those spikes are due to the inability of the generator to send all the PDUs in a very short time interval, and reordering of PDUs based on priorities does not occur at the generator, but at the router and satellite queues. Yet, even if the generator is conceived as including an output queue, where reordering can take place, all the high priority PDUs will be serviced first in a FIFO fashion, still creating the negative spikes. However, if simulator must respond to higher priority PDUs more quickly then a benefit is seen because their travel time is reduced.

One statistics that was affected by the introduction of HoL is the total travel time (latency) of the PDUs. Figure 69 compares the latency of PDUs bundled by Packet Alloying at 64 Kbps when the HoL strategy is in effect with the latency without using it. Under HoL the high priority PDUs arrived much faster than those not using the strategy at expenses of the low priority PDUs which are excessively

delayed. The zoom in picture shows that in the time interval [1900, 2450] seconds no PDUs originated at the ground station reached plane 7 in less than 5 seconds, while many ones did it in less than 1 second under HoL.

Table 8 compares the travel times for all the PDU bundles received at node 0 in plane 7 (site 21) with and without HoL, for combinations of 64 Kbps, 128 Kbps and 256 Kbps in the wireless links, classified by priority. The column *No HoL* ignores the priority information. The table shows that at 64 Kbps, 18,050 bundles representing 61 % of the blocks corresponding to priorities 5, 7 and 9 were received in less than 0.5 seconds on average, at expenses of the other 39 % that waited more than 25 seconds. How well these results conform to an acceptable simulation in terms of OTB fidelity is not known at this point as described in the Future Work section. It is possible that many of the low priority PDUs are redundant or unnecessary, as pointed out in [CD96, BCL97]. Yet, not using HoL will definitely affect the fidelity as all the PDUs are over 3.495 seconds of average travel time. At higher bandwidths the difference between using or not HoL becomes smaller, but in relative terms it is still important. For example, at 256 Kbps, the low priority PDUs are delayed by extra 2 milliseconds (from 0.264 sec. to 0.266 sec.) that represents an increment of 0.76 %, while the travel time of the high priority PDUs was decremented by 3 milliseconds, or 2.31 %.

Table 8 shows different totals of PDU bundles. This is probably due to different number of collisions, which are more common at lower bandwidths. Another observation is related to the average times less than 0.25 seconds. This average includes PDUs originated at the CONUS ground station as well as at other airplanes. Because the satellite queue is causing the main delay in the simulation, it is worth to examine the PDUs originated at the ground station only. Table 9 includes such PDU bundles only.



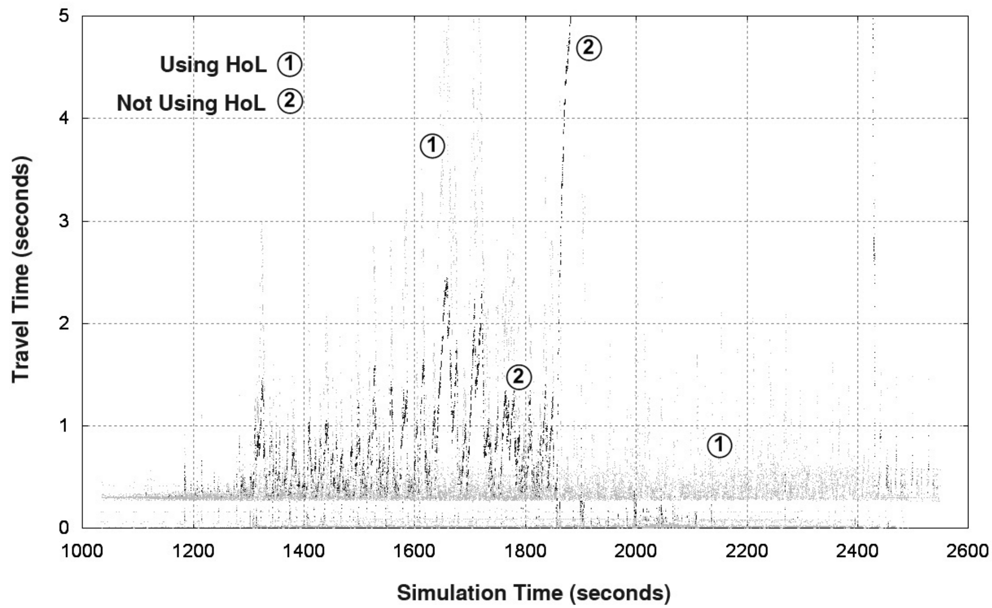


Figure 69: Effect of HoL on the Travel Time at Plane 7 (64 Kbps). The second graph is a zoom in where the lighter pixels correspond to the HoL strategy.

Table 8: Travel Time for all PDUs Received at Plane 7 by Priority (64 Kbps, 128 Kbps, 256 Kbps)

		No HoL		HoL	
Bandwidth	Priority	# PDU Bundles	Average (seconds)	# PDU Bundles	Average (seconds)
64 Kbps:	1	11,463	7.787	11,752	25.646
	5	8,291	9.538	8,218	0.405
	7	7,864	6.789	7,738	0.243
	9	1,892	3.495	1,882	0.176
	Total:	29,510		29,590	
128 Kbps:	1	13,640	0.299	13,649	0.319
	5	9,363	0.275	9,360	0.270
	7	8,842	0.211	8,842	0.199
	9	2,089	0.159	2,085	0.142
	Total:	33,934		33,936	
256 Kbps:	1	14,893	0.264	14,886	0.266
	5	10,161	0.237	10,163	0.238
	7	9,503	0.182	9,516	0.181
	9	2,181	0.130	2,189	0.127
	Total:	36,738		36,754	

Table 9: Travel Time for PDUs Originated at Ground Station and Received at Plane 7 by Priority (64 Kbps, 128 Kbps, 256 Kbps)

		No HoL		HoL	
Bandwidth	Priority	# PDUs Bundles	Average (seconds)	# PDUs	Average (seconds)
64 Kbps:	1	10,993	8.119	11,284	26.709
	5	7,009	11.273	6,938	0.467
	7	4,594	11.598	4,467	0.398
	9	798	8.198	782	0.349
	Total:	23,394		23,471	
128 Kbps:	1	13,139	0.309	13,149	0.330
	5	7,907	0.320	7,906	0.314
	7	5,368	0.337	5,372	0.318
	9	901	0.328	897	0.296
	Total:	27,315		27,324	
256 Kbps:	1	14,375	0.274	14,368	0.276
	5	8,603	0.278	8,605	0.278
	7	5,927	0.287	5,941	0.285
	9	945	0.279	952	0.275
	Total:	29,850		29,866	

By filtering out the PDUs from other airplanes, it is seen in Table 9 that all the average travel times are over 0.25 seconds. The table is not much different form the previous one and the HoL effect is still considerable. For example, at 128 Kbps we see that paying an increment of 6.8 % in the delay of low priority ESPDUs produces a decrement of 1.9 %, 5.6 % and 9.8 % in the latency of PDUs with priorities 5, 7 and 9, respectively. The frequently occurrence of PDUs scheduled at the same time during negative spikes suggests the urgency of those PDUs, indicating that a PDU priority scheme is worth of consideration.

CHAPTER 6

TRAFFIC OPTIMIZATION USING PACKET ALLOYING

The analysis of negative spikes in Section 5.6.6 motivated the concept of a possible solution to eliminate or reduce them by means of aggregating the participating PDUs. In order to do so, the PDUs were examined in more detail, looking for similarities and redundancies in their fields in order to formulate an aggregation strategy. Each type of PDU has its own internal structure made up of fields and values of different sizes. A study of all the logged PDUs in the MR1 vignette indicated that if two PDUs are of the same type and length then they have identical field structures, as indicated in Section 4.4. This is a key point in the proposed bundling algorithm.

Another observation from the logged PDUs is the fact that OTB schedules some sequences of consecutive PDUs using exactly the same timestamp, as in the sample sequence shown in Figure 58. This causes a bottleneck in generators due to the infeasibility of sending several packets at the same time. In most cases, consecutive PDUs of equal type and length differed in the contents of a few fields, presenting the possibility of merging them into a single PDU. The OMNeT simulation of the MR1 vignette using Packet Alloying will be referred to as Simulation MR1PA.

6.1 Input Data Logs in Simulation MR1PA

The summary PDU files described in Figure 20 of Section 5.2 contain 4 characters at the end of each PDU. The four characters are combinations of `S` standing for *send*, and `W` standing for *wait*. As described below, they provide information about the action to follow after processing each PDU. Six algorithms to predict that action are proposed and studied. They can be classified in two groups: *online algorithms*, which decide the next action based only on the already processed PDUs, and *offline algorithms*, which have access to all the past and future sequences of PDUs in advance.

In this simulation the online algorithms are *Neural-Network* prediction (see details on Section 4.3.2.2), *Always-Wait*, and *Always-Send*. The offline strategies are *Type*, *Type-Length*, and *Type-Length-Time*, which have the capability of *ideal prediction* due to their knowledge of the future. The assignment of OTB sites to computer nodes in Simulation MR1PA is the same as in Simulation MR1GS, and can be found in Section 5.7.1.

6.2 Slack Time Analysis

Figure 70 shows the slack time of the generator at the CONUS ground station for different predictive algorithms. The graph was created assigning 64 Kbps to all the wireless links and 100 milliseconds to the timeout period. As seen in the diagram, up to the second 1,600, all of the algorithms behaved alike, but around that point negative slack started to build up. The *Always-Send* algorithm, which is equivalent to the *non-bundling* algorithm used in Simulation MR1GS (see Figure 60), incurred in the largest negative slack, followed by a *Type-Length-Time* strategy. The neural

network approach performed relatively well, considering that its predictions are not perfectly accurate. The other algorithms are among the best in this simulation, and a close-up of their performance is shown in Figure 71.

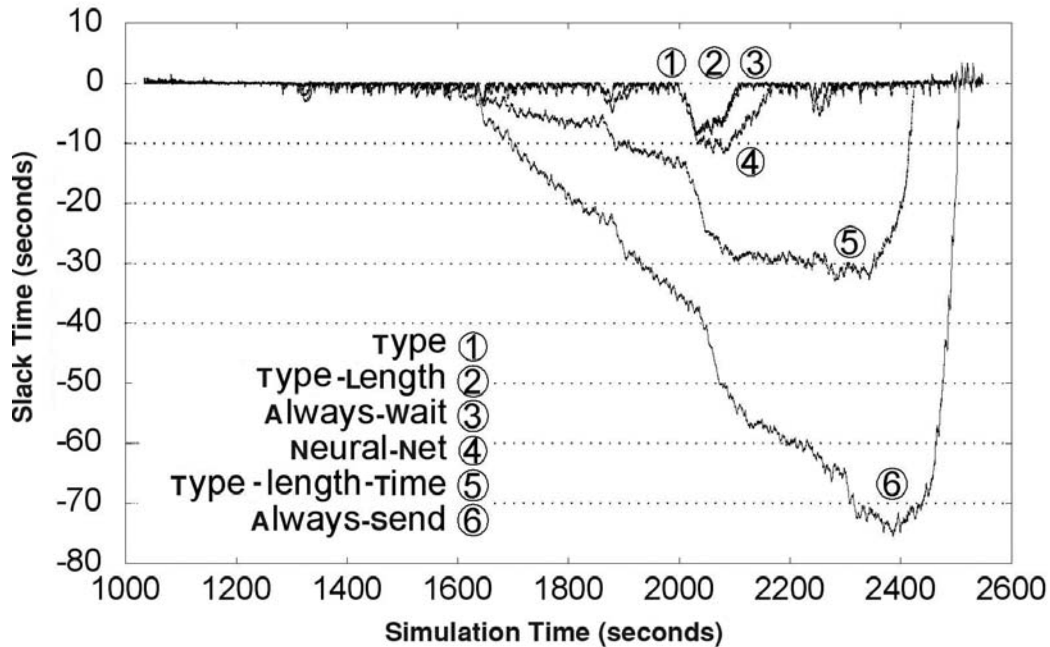


Figure 70: Slack Time at Ground Station for the 6 Predictive Strategies (64 Kbps)

From the graph in Figure 71, it can be concluded that the neural network approach could be improved by using a better learning mechanism and/or neural network architecture. The neural network algorithm predicts the PDU type based only on the time series of the past 44 PDU types. Therefore, its performance can be compared against the optimal *Type* algorithm, obtaining its competitive ratio, as defined in [FL02], for the cost function *negative slack time*, which resulted in $c = 3.75$ as indicated in Section 6.3.

Another observation from Figure 71 is that the decision of sending the current bundle based solely on the upcoming PDU type, performs as well as the one that considers the type and the length of each PDU. Therefore, a neural network approach

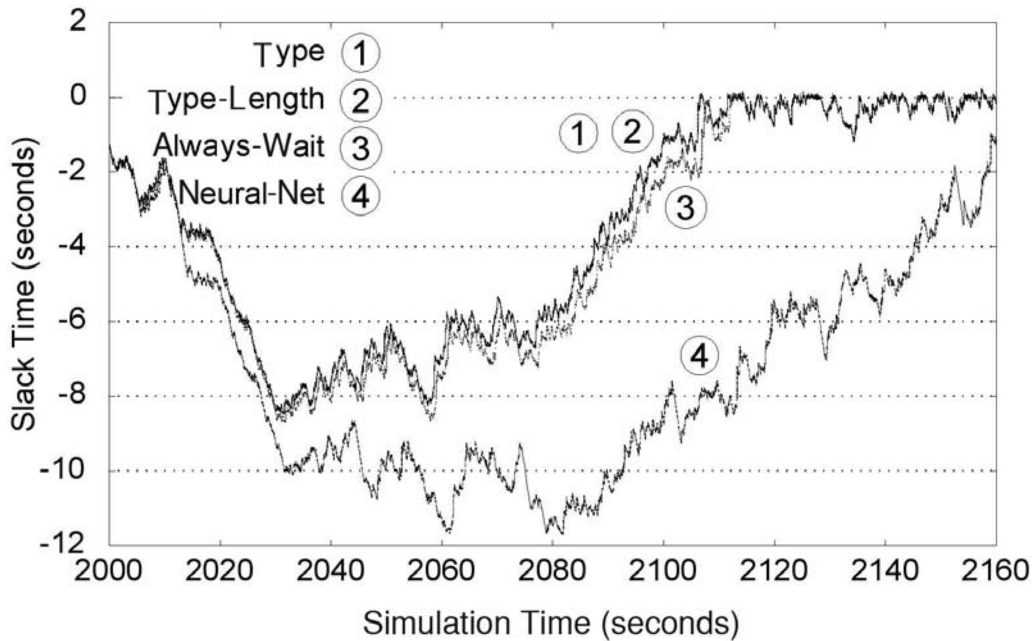


Figure 71: Comparison of Negative Slack for the Four Best Algorithms (64 Kbps)

could benefit from this observation by concentrating the effort in predicting the type only, instead of the type and the length.

However, the most interesting observation comes from the fact that the *Always-Wait* algorithm is almost as good as the one based on the *Type-Length*, and of course, *Always-Wait* is the simplest of all the strategies. The reason is that there is a high probability that the prediction based solely on the type agrees with the prediction based on the type and length. For example, an offline examination of the PDUs indicated that from the 50,230 PDUs sent by the CONUS ground station, 42,911 (85.4 %) implied the same action (wait or send) for both algorithms.

Table 10 shows the slack time average and standard deviation for all combinations of algorithms and bandwidths measured at the ground station. The average is a signed number; therefore, the larger the average is, the better the algorithm

performs. The average was calculated considering all the PDUs generated during the simulation.

Table 10: Slack Time Average and Standard Deviation for All the Studied Algorithms and Bandwidth Combinations Measured at the Ground Station. Best offline and online values are underlined

Average Std. Deviation	64 Kbps	128 Kbps	256 Kbps	512 Kbps
Type	<u>-0.758</u> 1.600	<u>-0.017</u> 0.109	<u>0.015</u> 0.073	<u>0.024</u> 0.066
Type- Length	-0.760 1.601	-0.018 0.110	0.015 0.073	0.024 0.066
Type-Length -Timestamp	-10.659 11.711	-0.027 0.115	0.013 0.073	0.023 0.066
Always- Wait	<u>-0.802</u> 1.689	<u>-0.017</u> 0.109	<u>0.016</u> 0.073	<u>0.024</u> 0.066
Neural- Network	-1.579 2.638	-0.044 0.162	0.008 0.085	0.022 0.069
Always- Send	-26.181 26.033	-0.054 0.176	0.006 0.085	0.021 0.069

From this table it is concluded that the *Always-Send* is the worst of the six algorithms, and *Always-Wait* is among the best. Because, *Always-Send* corresponds to the non-bundling option, it is clear that the type of bundling proposed here is advantageous compared to the DIS protocol.

Another observation comes from the fact that at 64 Kbps and 128 Kbps, the average slack time was negative for all the algorithms, but for 256 Kbps and above it is positive. A negative average indicates that the corresponding bandwidth is insufficient to handle the PDU traffic. Therefore, for the MR1 vignette, the wireless bandwidth should be at least 256 Kbps according to Table 10.

6.3 Travel Time Analysis

To enable analysis, each bundle sent includes the current time (T_{send}) attached with it, allowing the destinations to calculate the travel time T_{trav}, as indicated in Equation 5.5. Figure 72 shows the travel time measured at sink 0 onboard plane 0, for the *Always-Wait* strategy, using 64 Kbps and 128 Kbps in wireless links. It is clear from the graph that 64 Kbps is not sufficient to handle all the traffic required by the simulation, even with bundling. As seen, during the interval from second 2000 to second 2400 many of the PDUs took almost 40 seconds to arrive at their destinations, exceeding the fidelity requirements of the OTB simulation. However, a big improvement is obtained just by duplicating the bandwidth. At 128 Kbps, the latency was close to 0.8 seconds, as observed in Figure 73.

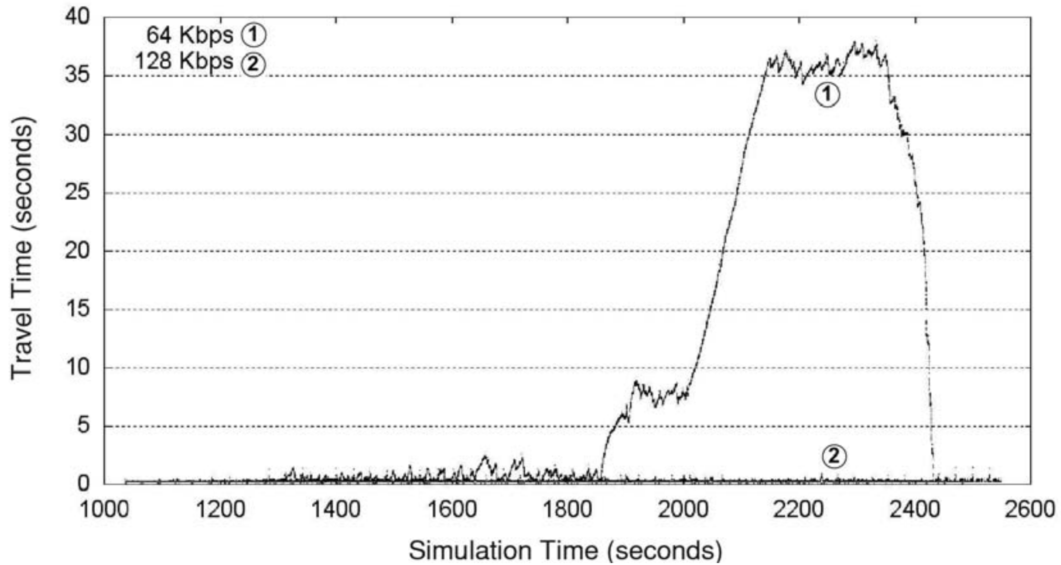


Figure 72: Travel Time for the *Always-Wait* Strategy, at Sink 0 in Plane 0 (64 Kbps and 128 Kbps)

Figure 73 shows that most of the PDUs take less than 0.4 seconds to reach their destinations. It is interesting to note the large concentration of PDUs near 0.25 seconds, which is the propagation delay for satellite signals. The graph also shows that some PDUs take less than 0.1 seconds of travel time. Those PDUs correspond to messages sent from other airplanes without passing through the satellite.

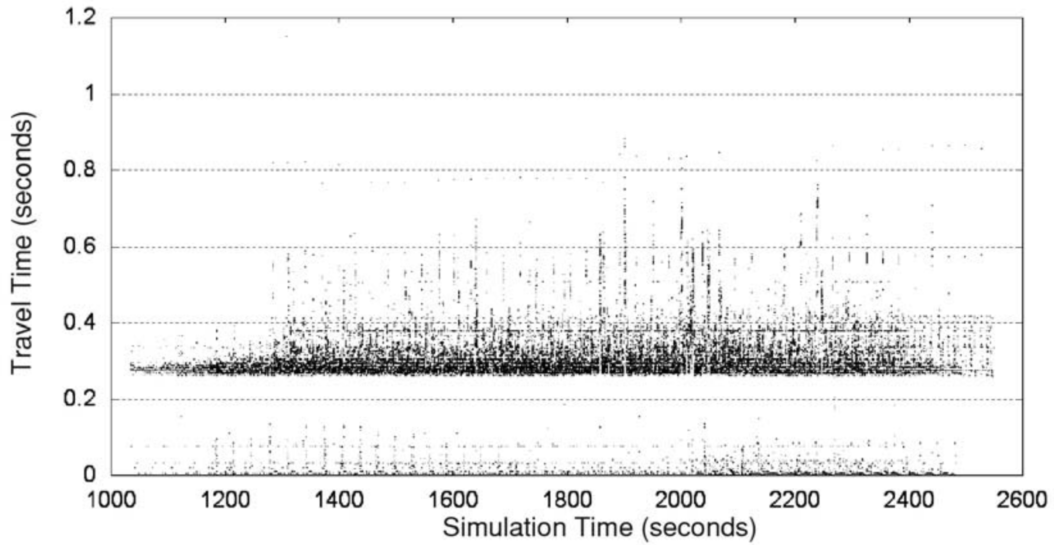


Figure 73: Close-up of Travel Time at Sink 0 in Plane 0 (128 Kbps)

Table 11 shows the average and standard deviation of the travel time for each combination of algorithm and bandwidth, measured at sink 0 onboard plane 0. Considering that approximately 83 % of the PDU traffic arriving at sink 0 comes from the ground station via satellite, and that for those PDUs, 0.255 seconds is an unavoidable delay, the table shows a very good behavior of the algorithms at 256 Kbps or more, giving a slight advantage to *Always-Wait* and *Neural-Network* over *Always-Send*.

Table 12 shows the total travel time for all the PDU bundles that arrived at node 0 in plane 7 (sink 21). The sum of all the travel times is an example of a cost

Table 11: Average and Standard Deviation of Travel Time Measured at Sink 0

Avgerage Std. Deviation	64 Kbps	128 Kbps	256 Kbps	512 Kbps
Type	<u>9.20</u>	0.304	0.262	0.249
	13.2	0.099	0.069	0.064
Type-	9.24	0.306	0.262	0.249
Length	13.2	0.101	0.069	0.064
Always-	9.43	<u>0.303</u>	<u>0.261</u>	0.249
Wait	13.5	0.099	0.069	0.064
Neural-	28.7	0.314	<u>0.261</u>	<u>0.248</u>
Network	33.2	0.119	0.069	0.064
Always-	64.0	0.333	0.263	0.251
Send	58.0	0.153	0.062	0.057

function that can be used to estimate the constant c for the c-competitiveness of the online algorithms, as defined in Section 4.2. According to Table 12, at 64 Kbps the best offline algorithm is *Type-Length*. Based on it, *Neural-Network* would have $c = 3.75$ and *Always-Wait* would have $c = 1.03$. However, we cannot assume that *Type-Length* is the optimal offline algorithm, and we would need to calculate the cost function for a large sample of simulation vignettes, as required by definitions 4.4 and 4.5. In fact, *Type-Length* can be improved in the following way. After processing a given PDU, if *Type-Length* predicts `W` (wait) but the next PDU will arrive after the timeout of the current bundle, then the waiting time would have been wasted. A better online algorithm could have analyzed this case and predict `S` (send).

At 256 Kbps, the *Type* strategy appears better than *Type-Length*, and the online algorithm *Always-Wait* results the best of all. This information is contradictory, and we explain it by saying that there is a better offline algorithm that overcomes the ones in the table. Nevertheless, a conclusion drawn from the table is that at higher bandwidths the differences between the different algorithms become smaller. For instance, at 256 Kbps *Neural-Network* has a $c = 1.15$ based on *Type-Length*, instead of the previous value of 3.75.

Table 12: Total Travel time at sink 21 (64 Kbps, 256 Kbps)

Bandwidth	Strategy	Total Travel Time (seconds)
64 Kbps	Type	222,589.264
	Type-Length	222,357.200
	Always-Wait	228,350.418
	Neural-Network	832,881.794
256 Kbps	Type	8,419.056
	Type-Length	8,431.477
	Always-Wait	8,357.483
	Neural-Network	9,725.795

6.4 Queue Length Analysis

Due to the nature of the PDU traffic in the simulation, two queues to focus attention on are the router queue onboard any aircraft, for instance on airplane 0, and the satellite queue. Figure 74 shows the satellite queue at 64 Kbps and 128 Kbps. It is clear from the graph that 64 Kbps is an insufficient bandwidth, causing the satellite queue to grow unbounded once it becomes full. The reason for having a descent after reaching a maximum of about 6,000 messages, is that the simulation is approaching its end and no more messages are sent from the generators. However, at 128 Kbps a significant change in the queue length is produced, keeping it at reasonably low values.

Another observation is that at 64 Kbps the graph does not reach zero at the end. This occurs because the queue status is reported only if another message enters the queue. After the arrival of the last message to the queue, the messages are consumed without being reported.

Table 13 displays the average and standard deviation of the satellite queue length for combinations of different algorithms and bandwidths. For transmissions clearly exceeding the channel capacity available, a *Type* strategy is shown to perform best,

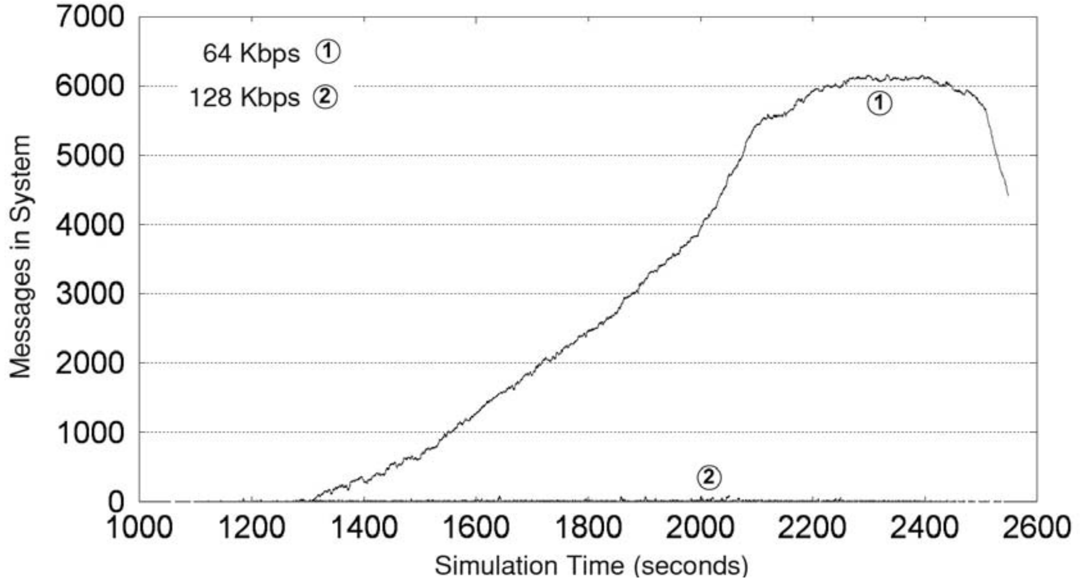


Figure 74: Messages in Satellite Showing the Impact of a Higher Bandwidth on its Queue (64 Kbps and 128 Kbps)

resulting in a 89.3 % improvement compared to an *Always-Send* strategy used by DIS. However, when the channel capacity is near to that of the demanded rate then it is seen that a single *Always-Wait* strategy can perform just as well, yielding a 30.3 % improvement over DIS. When the bandwidth is low, however, a *Type* strategy can outperform an *Always-Wait* strategy by 3.1 % as shown in for 64 Kbps in Table 13. These results are not surprising because *Type* is an offline algorithm, and good offline algorithms should outperform the online ones.

Among the studied online algorithms, the closest one to *Type* is *Neural-Network* that strives to predict the type of the next PDU in the sequence. Assuming that *Neural-Network* could be improved sufficiently to resemble the performance of *Type*, and defining the coefficient γ as the ratio of the channel capacity to the average bandwidth demand:

$$\gamma = \frac{channel\ capacity}{average\ bandwidth\ demand} \tag{6.1}$$

Table 13: Average and Standard Deviation in the Satellite Queue Length for Combinations of Algorithm and Bandwidth

Average: Std. Deviation:	64 Kbps	128 Kbps	256 Kbps	512 Kbps
type	316.97	2.38	0.91	0.56
	411.43	3.97	1.72	1.23
Type-Length	318.154	2.44	0.92	0.56
	412.273	4.13	1.75	1.26
Always-Wait	327.278	2.30	0.85	0.49
	421.161	3.88	1.69	1.16
Neural Network	1,028.47	3.58	1.24	0.79
	1,045.26	6.37	2.18	1.52
Always-Send	2,962.94	5.40	1.22	0.63
	2,236.83	10.78	2.55	1.57

then the decision tree in Figure 75 can be used to select the preferred PDU bundling strategy in each case.

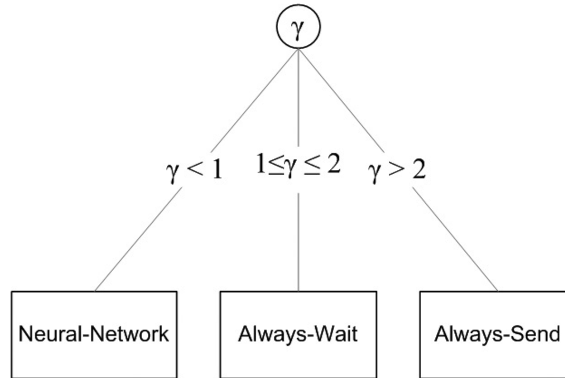


Figure 75: Preferred PDU Bundling Strategy

For low values of γ the demanded bandwidth is larger than the channel capacity. Type is the best offline algorithm in this case, but because it is offline, an improved Neural-Network is selected. If γ is somewhat larger than 1, for instance between 1 and 2, the channel capacity is sufficient to handle the traffic on the average, but there could be spikes of high demand. Always-Wait is the best choice in this scenario. When γ is large, for instance larger than 2, there is an excess of bandwidth as

compared to the demand, and alloying is not justified. Alloying implies the addition of a small delay while the algorithm is waiting for the next PDU. Always-Send is a good choice in this case because it is the simpler strategy, does not incur in extra delays and provides good performance.

6.5 Collision Accumulation

Collision accumulation in plane 7 at different bandwidth rates is given in Figure 76. The results from the simulation indicate that at 64 Kbps the highest collision rate measured at the router aboard airplane 7 was close to 12 collisions per second, and this occurred during the time interval [2050, 2100] in the WSP link that connects the satellite to the planes. At 64 Kbps, fewer than 4,800 collisions were detected in total for the *Always-Send* algorithm, which represents less than 8 % of the total number of PDUs. On the other hand, at 256 Kbps the total number of collisions for the *Always-Wait* algorithm was close to 2,100, or 5.3 % of all the bundles.

As Figure 76 shows, at 128 Kbps and 256 Kbps there is roughly a total difference of 1,000 fewer collisions for the *Always-Wait* than for the *Always-Send* algorithm. This indicates that bundling significantly reduces the number of collisions, given the same bandwidth for both algorithms. In addition, it can be noted that as the bandwidth increases, the number of collisions decreases, which is intuitively explained because at higher bandwidths the packets take less transmission time, and so the probability of a collision gets lower.

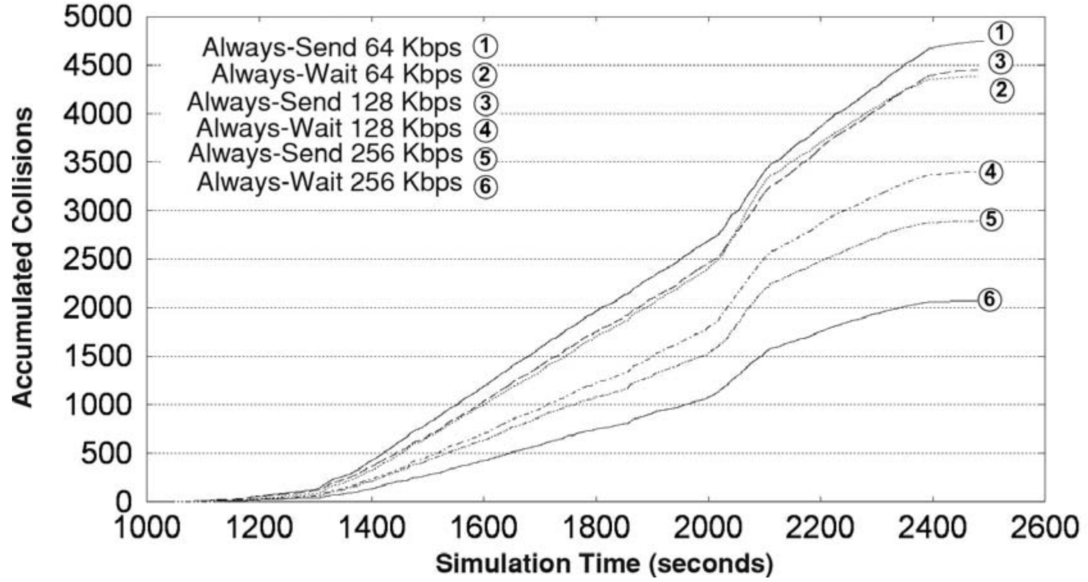


Figure 76: Collision Accumulation at Plane 7 (64, 256, 512, 1,024 Kbps)

6.6 Conclusions of Packet Alloying Simulation

The main conclusion of this simulation is that the type of aggregation proposed in this dissertation proved to be successful for DIS PDU transmission. Although their performance can vary with respect to each other, all the algorithms utilizing the proposed bundling strategies performed significantly better than the non-bundling *Always-Send* algorithm.

Furthermore, prediction based solely on the PDU type is almost as good as the prediction based on the type and length, and these predictions are better than the *Always-Wait* algorithm by half a second in some cases. Therefore, a neural network approach could be useful if the percentage of successful guesses is sufficiently high enough so that it outperforms the *Type* or *Always-Wait* algorithms.

Another conclusion is that the *Always-Wait* algorithm, although not optimal, gives very good results that are acceptable in many cases, especially if the band-

width is incremented. Taking into account that the shown results are simulations at 64 Kbps, at higher bandwidths the difference between *Always-Wait* and the perfect guessing algorithms (*Type*, *Type-Length*) becomes smaller, giving the more straightforward *Always-Wait* strategy more relevance. The final conclusion about the bandwidth is that 256 Kbps in wireless channels is the minimum bandwidth for the MR1 vignette that is sufficient by all metrics (slack time, travel time, queue length, and collisions).

The results in this Chapter demonstrate that the DIS traffic generated by OTB can be substantially reduced by the application of several techniques. PDU bundling techniques can diminish the negative spikes in the slack time during traffic generation. Inter and intra PDU redundancy can be eliminated by bundling and compression techniques.

CHAPTER 7

CONCLUSION

The subject of the dissertation is the data transmission scheduling for distributed simulations and the assessment of the required bandwidth, given the traffic specifications in the form of logged packets from an actual simulation. The background of this field of research was presented in Chapter 1 and Chapter 2. A drawback in the DIS protocol is its high network bandwidth requirements and the large computational loads placed on the host computers. Several attempts have been made to overcome the bandwidth problem. The general idea relies on finding new methods to reduce the network traffic. The methods include bundling and aggregation of packets, delta-PDU encoding, latency compensation, dead-reckoning algorithms, lossless data compression techniques, TCP/IP header compression, packet rescheduling, multicast routing, and packet transmission using priorities. In this dissertation we proposed a new method of the PDU alloying based on the internal structure of the packets, which has not been studied before.

There are several sets of conclusions that can be drawn. The first set corresponds to conclusions about the simulations performed in this dissertation. The second set corresponds to conclusions about the architecture-independent analysis proposed as a first approach to assess the bandwidth. The third set corresponds to conclusions about the OTB traffic analysis and excessive redundancy detected in the PDUs. The fourth set corresponds to conclusions about the effectiveness of the Packet Alloying

technique proposed, and the last set is about the importance of using HoL priority service in OTB simulations.

It is important to mention here that through this research the project team of Electrical and Computer Engineering Department of the University of Central Florida was familiarized with the OMNeT software, which is a tool in the public domain and provides a quality simulation environment for C++ programmers. The development of the simulator for handling PDUs of an OTB application is an example of OMNeT usage that can serve as a starting basis for other projects.

7.1 Summary of Accomplishments

Communication bandwidth and latency reduction techniques were developed for DIS protocols. Using logs from vignettes simulated by OTB, a discrete event simulator was developed to analyze PDU traffic over a wireless flying LAN. Alternative PDU bundling and compression techniques were studied under various metrics including *slack time*, *travel time*, *queue lengths*, and *collisions*. Based on these results, *Packet Alloying*, a technique for the bundling of transmitted packets, was proposed and evaluated.

The contributions of this dissertation include the formalization of the algorithm for Packet Alloying, the formalization of an independent analysis to estimate the bandwidth without using simulation, the proposal and study of different algorithms that predict the action (*Wait* or *Send*) in the packet generator before the next PDU becomes known, and the study of the effect of applying priority-based optimization using HoL service.

7.2 Simulations Performed

A total of six sets of simulations were performed, using 3 different vignettes and different assignment of OTB sites to simulation nodes as described below.

The first set of studies corresponds to Simulation ST: a vignette with Single Transmitter. It is a straightforward vignette that allowed the testing of the discrete event simulation concept. According to the results of the simulator and of the independent analysis, 64 Kbps in the wireless links is sufficient to handle the traffic.

The second set corresponds to Simulation DT: a vignette with Dual Transmitters. As predicted in the architecture-independent analysis, 64 Kbps in the wireless channels is insufficient in this simulation due to a large demand of bandwidth concentrated in a short time interval. Except for that spike, 64 Kbps can handle the remaining traffic.

The remaining four sets of simulations are based on the MR1 Vignette that contains six transmitters, and one of them, the main transmitter, produces the majority (83 %) of the PDUs. The third set was gathered from 6 transmitters and named Simulation MR1T6 and the main transmitter was assigned to an node onboard an airplane. The architecture-independent analysis predicted that 200 Kbps would be sufficient to handle the traffic, but because this is not a standard bandwidth, the conclusions considers 256 Kbps as the minimum value, which agrees with the simulation results.

The fourth set of simulations is called Simulation MR1GS in which case the main transmitter was assigned to the ground station. The analysis of the slack time to transmit the next PDU revealed the occurrence of negative slack spikes at regular time intervals. The studied PDUs participating in those negative spikes showed that they constituted sequences scheduled at the same time or almost the same time, and usually of the same type and length. The structure of such PDUs

was investigated, concluding that they were similar in structure. The three main types of PDUs found in all negative spikes were: `po_fire_parameters`, `po_line` and `po_task_state` PDUs. The `po_fire_parameters` PDUs are the main sequence responsible for negative spikes, perhaps because the generating entity (ground station) fires bursts against some enemy. This hypothesis needs to be corroborated against an actual run of the OTB vignette to conclude that firing activities are the main cause of negative spikes. The phenomenon triggered the idea of bundling those PDUs to remove redundant fields. This idea was the base of the proposed Packet Alloying bundling. Also, the idea of assigning priorities to them was considered and studied in the HoL simulation.

One problem with the logged data of the MR1 Vignette was that the OTB simulation was run on a network of six computers, using virtual sites for the participating entities. The effect of this scenario is that the messages sent from one site to another did not travel physical distances like the satellite link and did not waited on router queues, etc. The acknowledgments corresponding to delivered PDUs were received in a fraction of the expected time, causing OTB to send the next PDUs more frequently than in the real scenario. In other words, all the timestamps in the PDUs correspond to a much faster network. The OMNeT simulator used those timestamps unchanged, which could explain in part the reason of negative spikes.

The OTB simulation could be improved by using a variety of different computers for different sites, and one computer for simulating the transmission and propagation latency in the network. This environment will certainly produce logged PDUs of more realistic quality.

The fifth simulation studied the Packet Alloying for the MR1 Vignette keeping the same assignment of Simulation MR1GS. The effect of bundling was significant, as indicated in the following examples. Applying *Always-Wait* to the MR1 vignette setting the wireless links to 64 Kbps, a reduction in the magnitude of negative slack

time from -75 to -9 seconds for the worst spike was achieved, which represents a reduction of 88 %. Similarly, at 64 Kbps, *Always-Wait* reduced the average satellite queue length from 2,963 to 327 messages for a 89 % reduction.

The last simulation studied the effect of applying HoL priority service to the satellite and router queues, using priorities based on the PDUs that were found more frequently in negative spikes of the slack time. The results were successful in the sense that PDUs of highest priority arrived earlier to the destinations at the expense of lower priority PDUs that arrived with high delays.

7.3 Architecture-Independent Analysis

The architecture-independent analysis performed on the logged data is an important procedure in the assessment of bandwidth because it gives a good initial insight about the minimum instantaneous bandwidth required at a fraction of the cost of a complete simulation. It can be used also to identify periods of low and high network traffic, and correlate them with actions being developed by the simulated parties for a better understanding of the simulation behavior. In all the simulations performed, the bandwidth predicted by the independent analysis agreed with the bandwidth calculated experimentally using the simulator. Proofs for the minimum local bandwidth and minimum average bandwidth were derived showing $\overline{B}_{i,j} = (\sum_{k=i}^{j-1} 8L_k)/(T_j - T_i - (j-i)g)$ and $B_{a,b} = \max_{a \leq k < b} \{\overline{B}_{k,b}\}$. We conclude that the independent analysis is a valuable procedure in bandwidth assessment, not requiring the development of a discrete event simulator.

7.4 Conclusions about OTB Traffic

Several characteristics of the Embedded Simulation traffic affect the bundling of PDUs. During the simulation, the participants interact with each other in real time. Therefore, most PDUs are constrained to be delivered within short time delays, usually less than one second. ES traffic contains 70 % or more of Entity State PDUs, which in some cases are redundant or not urgent. Also, considering that some PDUs can be rescheduled and sent in a different order without adversely affecting the overall simulation, we conclude that the assignment of priorities to PDUs should give better results, especially under low bandwidths.

Some high priority PDUs like fire and detonation occur in short bursts, and they are usually sent at the same time, creating negative slack spikes that operate against real time objectives. The main cause of those negatives spikes is the scheduling of PDUs having exactly the same timestamp. The analysis of the largest spikes showed that PDUs of type `po_fire_parameters` are the main components of the spikes and sequences of eight or more PDUs were commonly found. Comparisons of samples of `po_fire_parameters` PDUs for the same spike indicated that they are very similar in structure and content, having differences related to PDU identification and memory address of the PDU fields only. The magnitude of negative spikes was derived and proved to be $m = \max_{0 \leq j \leq k} \{m_{i+j}\}$, where $m_{i+j} = T_{i+j} - (T_i + j \cdot g + (\sum_{u=0}^{j-1} L_{i+u})/B)$.

Rescheduling of the PDUs is also a technique that can alleviate the occurrence of these negative spikes in the slack time. The rescheduling and transmission of PDUs is an attempt to reduce the negative slack spikes by transferring some of their PDUs to periods of positive slack. Not only the `po_fire_parameters` PDUs are subject to be rescheduled, but any PDU that involves some sort of negative slack could be rescheduled to obtain a traffic as close as possible to a burst-free model to keep the channels busy yet not flooded. A side effect of reducing the PDU traffic is

a decrease of collisions, especially if the PDUs are relatively long, as is the case of `po_fire_parameters` PDUs. Ten consecutive PDUs of this type account for 5,280 bytes plus the time gaps between frames. During the transmission time of these PDUs, the channels are heavily occupied and any other attempt to transmit from another station over the same channel will end up in a collision. By sending just one PDU of approximately 550 bytes will decrease the probability of having a collision.

The DIS protocol is responsible for the timestamps of the PDU packets. If many PDUs are scheduled not only at the same microsecond, but also within a very short time interval, the effect is similar to a negative spike. The experiments with the OMNeT simulation showed that mixed with negative spikes there are many positive ones. The positive spikes indicate that the channel is not used during those intervals and so rescheduling of the PDUs could bring a better utilization of the channel. If a sequence of PDUs timestamped at almost the same time followed by periods of rest can be recognized and predicted, then the scheduler could implement a type of scheduling to refrain from sending those PDUs so close to each other and spread the sending times evenly throughout the intervals of low traffic. In doing so, the length of each PDU must be taken into account because it is proportional to the transmission time. In this research a neural network was implemented to predict the type of the next PDU with success of almost 70 % of correct predictions out of 27 different possible PDU types.

PDUs contain high levels of redundancy, both inside each PDU and across PDUs. Bundling and compression are techniques that can eliminate the redundancy, lowering the bandwidth demand and reducing collisions. PDUs have a definite structure made up of fields of different sizes, that are determined by the type and length of the PDUs. This characteristic allowed the comparison of PDUs at the field level, facilitating the extraction of the differences, as implemented by Packet Alloying.

In the DIS protocol, PDUs are broadcasted. This characteristic simplifies the PDU header since a particular destination is not needed, facilitating also the bundling and routing process. It is easier to bundle several PDUs if all of them have a common destination than if they were sent to different places. However, broadcasting contributes to the proliferation of messages that might not be needed at some sites because of their far distance or other reason. Multicast is a better alternative already included in protocols like DIS-Lite and HLA. Bundling in multicast mode should consider the destination as part of the definition of compatibility between PDUs.

Bandwidth plays a key role in the performance of the OTB simulation, as shown by the following observations extracted from Simulation MR1GS. Just by increasing the bandwidth in wireless channels from 64 Kbps to 128 Kbps, the negative slack time changed from values close to -75 seconds to values near -1.5 seconds. Increasing the wireless bandwidth to 200 Kbps or more produces a significant change in the travel time, as all of the PDUs fall below the level of 0.5 seconds. At 64 Kbps the satellite queue becomes extremely long, reaching values over 6,000 messages. The improvement from 64 Kbps to 256 Kbps is significant, requiring queue storage for 35 messages only at the highest peak. At 64 Kbps the number of collisions represents approximately 8 % of the total number of PDUs, while at 256 Kbps the percentage descends to 5 %.

7.5 Packet Alloying

In this dissertation, the possibility was investigated of introducing a new aggregation strategy to eliminate redundancy in consecutive PDUs. For instance, if several `po_fire_parameters` PDUs are produced, only one physical PDU bundle is

actually sent, including the non-redundant fields of the bundled PDUs. This aggregation constitutes a lossless compression technique in which the extraction of the original PDUs occurs at the destinations in a straightforward manner. The new technique significantly lowered the queue lengths of routers and satellite, and decreased the travel times of PDUs at low bandwidths, as indicated above in Section 7.2.

Three online algorithms were proposed: *Neural-Network*, *Always-Wait* and *Always-Send*, as well as three offline algorithms: *Type*, *Type-Length* and *Type-Length-Time*. After analyzing the online algorithms, the main conclusion is that Always-Wait is one of the best algorithms for predicting the action to be performed after processing the current PDU. The possible actions are to *Wait* for the next PDU in an attempt to aggregate it with the current bundle, or to *Send* the current bundle starting a new bundle when the next PDU be delivered. The c-competitive index was estimated for the online algorithms based on the offline ones. Based on *Type-Length* at 64 Kbps, *Neural-Network* would have $c = 3.75$ and *Always-Wait* would have $c = 1.03$. However, we cannot assume that *Type-Length* is the optimal offline algorithm, and we would need to calculate the cost function for a large sample of simulation vignettes, as required by definitions 4.4 and 4.5. For *Always-Wait* the index was very close to 1. The conclusion of this observation is that the offline algorithms are not optimal, and an example of a possible improvement was given. Also, it can be concluded that the Neural-Network strategy can be improved, possibly by using more neurons, a longer input sequence and extended training sessions.

For transmissions clearly exceeding the channel capacity available, a *Type* strategy is shown to perform best, resulting in a 89.3 % improvement compared to an *Always-Send* strategy used by DIS. However, when the channel capacity is near to that of the demanded rate then it is seen that a single *Always-Wait* strategy can perform just as well, yielding a 30.3 % improvement over DIS. When the bandwidth

is low, however, a *Type* strategy can outperform an *Always-Wait* strategy by 3.1 % as shown in for 64 Kbps in Table 13. These results are not surprising because *Type* is an offline algorithm, and good offline algorithms should outperform the online ones. The decision tree in Figure 75 can be used to select the preferred PDU bundling strategy in each case. For low values of γ the demanded bandwidth is larger than the channel capacity. Type is the best offline algorithm in this case, but because it is offline, an improved Neural-Network is selected. If γ is somewhat larger than 1, for instance between 1 and 2, the channel capacity is sufficient to handle the traffic on the average, but there could be spikes of high demand. Always-Wait is the best choice in this scenario. When γ is large, for instance larger than 2, there is an excess of bandwidth as compared to the demand, and alloying is not justified. Alloying implies the addition of a small delay while the algorithm is waiting for the next PDU. Always-Send is a good choice in this case because it is the simpler strategy, does not incur in extra delays and provides good performance.

Some difficulties were assimilated during the research. OTB traffic contains PO_PDUs, which documentation is restricted. Due to this, the specific treatment of PO_PDUs and further incorporation in the OMNeT simulator during the development stage presented an unsolved disadvantage. The treatment of PO_PDUs was handled as if they were IEEE PDUs in the DIS protocol. But even in this case, the OTB logs were in a text format, not in binary. This caused difficulties during the bundling operation because text fields can be of different length than the corresponding binary ones, and comparisons are more cumbersome. The proposed alloying technique was kept at an abstract mathematical level. If more detailed information abut PO_PDUs had been available to the research, the technique could have reached the implementation level of PO_PDUs. For instance, the subindexes inside the bundled fields could have been specified in terms of bit length, content, and position within the PDU.

All of the statistics presented indicate that bundling is an effective technique for reducing the PDU traffic and better utilize the bandwidth. The reductions in negative slack (Figures 70 and 71), travel time (Table 11), satellite queue length (Table 13), and number of collisions (Figure 40) are all indicators in that sense, as well as the results mentioned above in Section 7.2.

The replication of PDUs through bundling presented in this research differs from other proposals [US95a, Tay95, Tay96b, Tay96a, Ful96, BCL97, PW98, WMS01] in several ways. First, bundling takes into account the internal structure of each PDU; only PDUs of the same type and length are put together in a bundle. Second, the resulting bundle has a structure similar to any other PDU and can be considered a PDU of a different type, subjectable to further bundling or compression technique if desired. Third, the bundling algorithm is straightforward to implement, as well as the extraction of individual PDUs at the destination. Each bundle is independent of the others and all the information needed to extract the PDUs is contained in the same bundle. An initial *reference* PDU sent to the destinations containing baseline values is not used in this approach.

7.6 HoL Priority Service

Simulations using HoL service indicate that assignment of priorities to PDUs have an impact on the OTB simulation that can be beneficial, especially when the satellite and router queues are long under heavy traffic loads. At the end of Chapter 5 the comparison given in Table 8 and Table 9 indicates that it is possible to delay less than half of the total PDUs, which are of low priority, in return of a large speedup of the others. At 64 Kbps, 18,050 bundles representing 61 % of the blocks corresponding to priorities 5, 7 and 9 were received in less than 0.5 seconds

on average, at expenses of the other 39 % that waited more than 25 seconds. It is possible that the assignment of priorities to PDUs given in that Chapter can be improved. It might be that a finer granularity could produce better results. In any case, only a man-in-the-loop simulation of the OTB vignette using HoL service can indicate if an assignment is worthwhile, but the results stated here suggest that a HoL strategy should be taken into consideration.

The complete impact of HoL cannot be measured by the OMNeT simulator alone because it does not interpret the PDU contents. Therefore, sending the high priority PDUs before those of lower priorities has an impact in the fidelity of the OTB simulation, but not in the OMNeT traffic simulation. The study of this impact is left as future work.

7.7 Future Work

The results in this dissertation can surely be a point of departure for further research. One possible branch for the continuation of this work is the obtention of a better Neural-Network predictive algorithm. The current one predicts the next PDU type with a certainty of near 70 %. With an alternate NN architecture or more training samples, this value could be elevated to 90 % or more. If an improved neural network is developed, it could outperform the *Always-Wait* strategy. However, a careful comparison in terms of simplicity and usage of CPU time and memory between both algorithms would be required.

Another possible continuation of the project deals with the incorporation of specific formats of PO_PDUs in the bundling strategies, instead of assuming the formats of IEEE PDUs. Otherwise the project cannot reach the implementation level for maximum benefit. Future research will require access to the specific formats

and characteristics of PO_PDUs. Also, the implicit assumption that in OTB some PDUs can be rescheduled and be sent in a different order without adversely affecting the overall simulation must be corroborated. It is known that only some PDU types can be sent out of sequence without impacting the casuality of the simulation.

We propose to identify better priority levels for PDUs, combining them with HoL queueing service. In this dissertation, the priorities were based on the occurrence of PDUs participating in a sample of negative spikes. It is possible that other criteria or the study of other spike samples can give better results to the simulation fidelity. In any case, the final results of the priority assignment have to be corroborated in a real OTB simulation.

In this dissertation standard compression techniques were not applied to the resulting bundles. It is proposed to study the combined effect of bundling with compression techniques, but real binary PDUs would be required for this project follow-on. Compression can be applied at two levels: data compression of the PDU data, and compression of the TCP/IP headers. Both compression techniques could be investigated.

Other possibilities of future work are open. The bundling algorithm could be extended to consider PDUs of the same type, but different length as candidates to be included in the bundle. Also, the bundling of PDUs of different types could also be researched as long as the resulting bundle still maintains the structure of a PDU. The implementation of the said PDU compression and rescheduling techniques could be made inside OTB directly, or by appending a filter to the OTB output, together with a de-filter module at the receiving site. The filter has the advantage of not modifying the current OTB implementation, at a cost of less than maximum efficiency.

An additional product obtainable from the project is the development and maintainability of UCF OMNeT++ models and logger files based on generic libraries

suitable for FCS rapid prototyping generation. A self-contained executable demo is a valuable help for future presentations of any simulation project.

The bundling strategy aimed at preserving the message structure could be applied to other communication protocols or data streams besides DIS. For instance, if a large database needs to be transmitted through a slow network and the records have some sequence relationship such that repeated fields are often found in consecutive records, the records could be bundled using the algorithms presented in this dissertation.

APPENDIX A

MR1 VIGNETTE

This vignette is due to Dr. Avelino González.

A.1 Background

In 2014, twenty years of independence for the Trans-Caucasus States found serious socio-political, ethno-religious, and economic conflict spreading throughout the region. Azerbaijan emerged as the leading economic power through the exploitation of Caspian and Central Asian oil reserves. Azerbaijan's politics were deeply divided; its citizens and Karabakh refugees demanded the government take military action against the Armenian Karabakh that forced them to flee. The Azerbaijani government refused to act, and refugees from the Nagorno-Karabakh Internal Liberation Organization (NKILO), using terror and armed force to achieve their goals, began a cross-border unconventional campaign designed to force a confrontation between the two countries. Observing these developments, Armenia and Iran viewed the Azerbaijani government's instability as an opportunity to expand their influence in the region for political gain. Armenia began massing maneuver forces along the Azerbaijani border and repositioned mobile Theater Ballistic Missile launchers. Both countries perceived a low risk of failure in executing their campaign strategy and were willing to impose a military solution upon *the Azerbaijani problem*.

In November 2014, initial reports of the Caspian Sea Peninsula crisis caused the U.S. to take steps to improve its awareness of the developing situation. The Secretary of Defense redirected intelligence assets to focus on the region and directed political and military planners to formulate contingency plans for U.S. engagement in the region. They determined an Army Objective Force Unit of Employment 2, operating as the Army component of a joint force, would be required to accomplish U.S. goals in the region and assigned operational control of the 15th Division air-ground task

force to USEUCOM for planning purposes. Warning orders were issued through USEUCOM to the U.S. 15th Division air-ground task force, and supporting attack and lift aviation assets to begin their own planning. U. S. Army Europe (USAREUR) and its theater support command (TSC) reviewed and updated contingency plans and refined the sustainment preparation of the theater. The TSC issued warning orders and created a provisional logistics/sustainment task organization called the Area Support Group (ASG) that would support land forces employed in theater.

In late November, the Azeri Islamic Brotherhood (AIB), a coalition of anti-government factions supported by NKILO and the Azerbaijani National Front for Revolutionary Action (ANFRA) military forces, subverted the bulk of an Azeri Motorized Rifle Brigade, which mutinied to realign with this faction. The brigade seized control of most of the historically significant Icheri Sheher (Inner Town) district in Baku. However, a desperate defense by loyal government forces managed to secure the centers of government within the capital city. Meanwhile, two armed clan-based factions of the Azeri Islamic Brotherhood, the Aziz and Daha, extended their control of the eastern and western outskirts of Baku, respectively, and intensified their efforts to overthrow the legitimate government. As a last resort, the Azerbaijani government requested assistance from the Russian Federation to defeat the insurgents and preclude an anticipated invasion by Armenian forces. On 15 December, Russia proposed a coalition of U.S. and Russian forces to restore order within Azerbaijan and stabilize the government. Two days later, the U.S. agreed to the proposal and the two nations created a coalition force and outlined its employment plan. The joint force commander, United States European Command (USEUCOM), and his Russian counterpart formed a coalition staff that included a coalition/joint theater logistics management element (C/JTLME). The C/JTLME continued to develop plans to logistically support coalition forces employed in theater and to determine the most efficient use of all coalition movement, sustainment, and facilities assets.

United States European Command focused its main effort at developing the situation and expanding the knowledge base already resident from the Operational Net Assessment of this region. They pre-positioned incremental force packages to establish a military presence in the region and deter any further hostilities, establishing a C4ISR architecture, and posturing to project forces directly into Azerbaijan and to dismantle Armenian C4ISR and fires systems. The combatant commander deployed Special Operations Forces (SOF) into the region, adding an additional layer of intelligence collection assets to the national-level space and air-based assets already operating over the region. Initially, their efforts were focused on developing the situation in the region of the beleaguered government in Baku. But as the 15th Division matured its plans, SOF teams shifted to provide coverage of the airfields the 15th Division planned to use as tactical points of entry for one brigade-sized Unit of Action (UA), the 1st Brigade UAs. The 1st Brigade UA is composed of three Battalions, the 1st, 2nd and 3rd.

A.2 General Vignette Description

This section describes the vignette in detail. It focuses on the 3rd battalion of the 1st Brigade Unit of Action. More specifically, it focuses on the lead element of the 3rd battalion - the Alpha Company. This company comes upon fortified defenses of the ANFRA forces and must destroy them to make way for the main element of the 3rd battalion coming up behind them. This is described in this section.

A.2.1 Situation and Mission Prior to Start of Vignette

The 1st and 2nd Battalions of the 1st Brigade UA are already on the ground before the beginning of this vignette. They have attacked the enemy forces in the city of Baku, defeated the subverted Azer brigade that controlled the City Center (referred to as the *Icheri Sheher Brigade*). Moreover, they confronted and routed the AIB forces in the vicinity of Baku. The 1st Battalion was subsequently tasked with pursuing the withdrawing AIB enemy forces retreating towards Agdam, and to continue on to Agdam and occupy it. The 2nd Battalion was ordered to maintain pressure on the Icheri Sheher Brigade in Baku to defeat it in detail.

In the meantime, 300 Km to the west, ANFRA forces, attacked across the Armenian border, seized the city of Agdam, and continued eastward to join with the retreating AIB forces and attempt to relieve the beleaguered Icheri Sheher Brigade in Baku. However, surprised by the rapid defeat of their allies in Baku, the ANFRA forces suddenly found themselves in an exposed position in the wide river valley between Agdam and Baku. Aware that the U. S. forces (the 1st Brigade UA) were mounting an operation to move westward to secure Agdam and restore the border, ANFRA forces began a delaying operation, designed to buy time for establishing a defense of Agdam while slowing and inflicting casualties on the attacking U. S. force. Keys to their hopes of success were preservation of the delaying force and effective use of target acquisition systems linked to long-range artillery systems.

The 3rd Battalion of the 1st UA Brigade now comes into the picture in this vignette with orders to attack and destroy the delaying forces of the ANFRA in order to permit the 2nd battalion to complete its mission of recapturing Agdam. The 3rd Battalion is in the midst of an airlift operation from a transfer point in Turkey when its specific mission is given to the commander. It must land, stage the

assets, organize itself and very rapidly move to accomplish its objective. Speed in this mission is of the essence.

A.2.2 The 1st UA Prepares for Entry Operations

The commander of the 3rd Battalion, on the way to the AOR via an airlift operation, was given a warning order to prepare to deploy immediately upon landing, and attack and destroy the delaying forces of the ANFRA. If successful, this would permit the 2nd Battalion of the 1st UA Brigade to complete its mission. The commander of the 1st UA used information from coalition/joint theater logistics management element (C/JTLME) fused with intelligence reporting from airborne assets and SOF teams operating in the area to select one airfield in vicinity of Baku (60 Km NW of the city) as his planned point of entry, as shown in Figure 1.

3.3 Mounted Formation Conducts Pursuit and Exploitation

Shortly after landing in their designated entry points, the 3rd Battalion of the 1st UA reorganizes and moves towards the ANFRA forces in open rolling terrain with some variance of complexity, such as defiles and small villages. The enemy is a combination of conventional forces, paramilitary, and special police challenging the UA forces with both direct military combat engagements and asymmetric means. The 3rd Battalion moves to contact with the ANFRA forces with the intent to maintain pressure on delaying forces, dislocate them, and force them into a battle while moving through open and rolling terrain so they could be destroyed by assault. To minimize his vulnerability to the enemy's long-range artillery systems, the commander planned to move his battalion dispersed on multiple axes while fighting an aggressive counter-reconnaissance effort. The result was near autonomous operations by each company, a common operating picture enhanced by situational

awareness and networked fires ensured the force remained interdependent and mutually supporting.

A.3 Specific Vignette for Project

As the 3rd Battalion of the 1st brigade UA advanced rapidly to meet the flank of the delaying force, the aviation detachment identified an enemy defensive position 60 Km in advance of the 3rd Battalion's lead elements (the Alpha company). The position was carefully selected by ANFRA forces to protect the AIB force withdrawing from Baku. The positions overlooked the best approaches to a river crossing along their line of withdrawal. Knowing that the lead Company (Alpha) would close on the reported location in just over an hour, the aviation unit used its sensors to identify specific target locations within the enemy position. Other sensors, mounted on unmanned aerial vehicles (UAV), were diverted from other areas to further develop the common operational picture. Their observations revealed that the position was well defended by a combination of dismounted infantry elements, Draega tanks, and Garm missile launchers in hastily prepared survivability positions. Minefields protecting the position from direct assault were still incomplete and operators of the advanced sensors on UAVs observing the area located several exploitable gaps and ensured they were portrayed on the common operational picture (COP). The scenario is depicted in Figure 77.

Quickly adapting his scheme of maneuver to the developing situation, the alpha company commander directed his reconnaissance assets to locate river crossing sites that were beyond the line of sight of the defensive position. When one was located north of the defensive position, the alpha company commander used his embed-

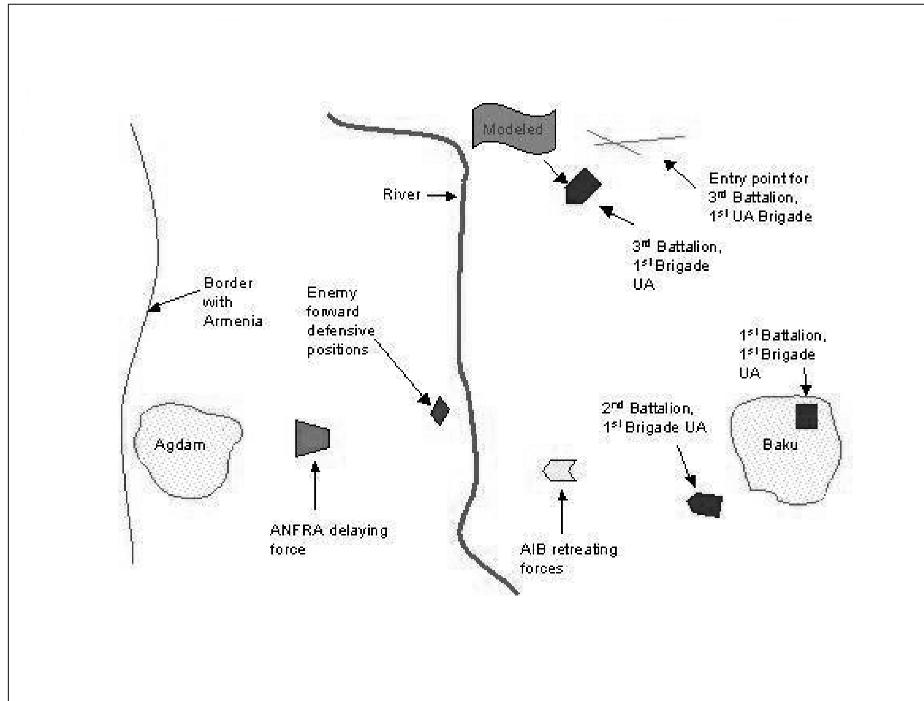


Figure 77: Overall View of Theater of Operations

ded collaborative planning tools to locate an ideal engagement area on routes the defenders would probably use as they were dislodged from their positions.

The Alpha Company commander directed the first platoon to cross north of the river and occupy positions that allowed them to place direct fires on defending forces as they entered the engagement area. Teamed with RAH-66 Comanches from the UA's aviation detachment, the 1st platoon brought the integrated fires of the UA's network to bear on the withdrawing forces.

The Second platoon was directed to cross the river some distance south of the defensive position and occupy positions that forced the withdrawing enemy towards the engagement area. The remaining two platoons were ordered to attack the enemy position and compel the defending forces to withdraw, enabling their defeat in detail. Still 30 km from the enemy position, the alpha company commander reviewed the continuously updated common operational picture (COP) for obstacles along his

intended axis of advance. While he watched, a newly identified minefield was posted on the display. Using the same planning tools, he quickly determined new routes for each of his platoons, directing them towards bypasses around the minefield, using line-of-sight evaluation tools to ensure the force stayed out of the enemy's line-of-sight as they maneuvered around the flank of the defending forces. Figure 78 provides detail about the target defensive positions as well as the crossing points for the 1st and 2nd platoons.

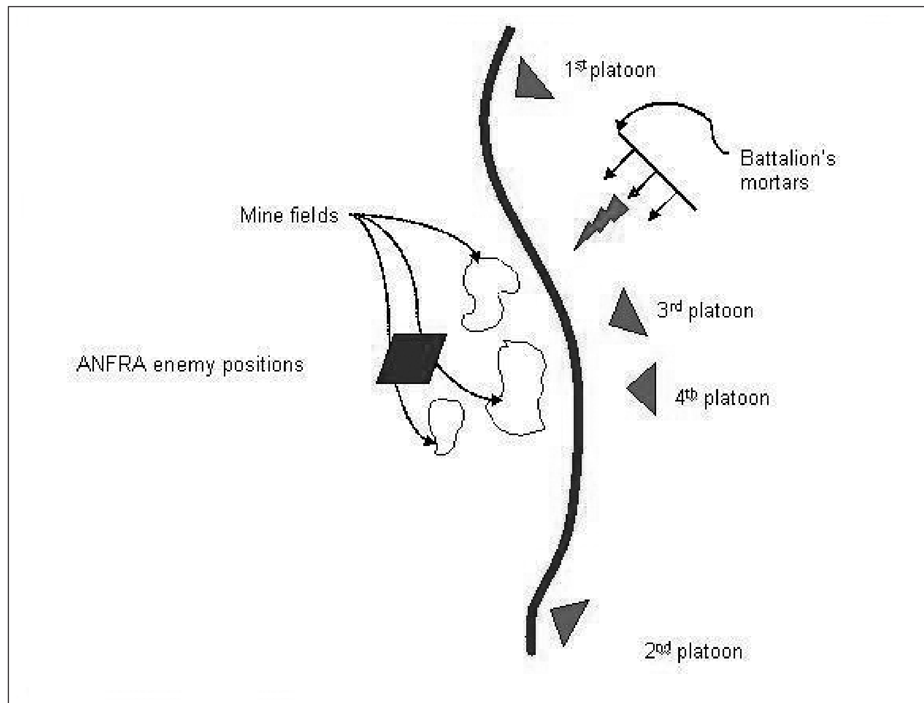


Figure 78: Details of Attack on Defensive Positions of ANFRA

When they closed to a range of 12 km, the alpha company's mortars began the attack on the defensive position. Pulling pin-point targeting data from the common operational picture, they delivered precision munitions aimed directly at the vehicles defiladed in the survivability positions within the enemy's defense. Their lethal, top-attack munitions quickly destroyed all but five vehicles.

Still too far away to directly observe the enemy positions, the Alpha Company commander used the split screen option on his display to watch both the map display of the common operational picture and live-video feed from the unmanned aerial vehicles observing the enemy's position. He watched as the five surviving vehicles, three Draega tanks and two Garm missile launchers, left their positions to flee towards Agdam, leaving the remaining dismounted defenders easy prey for the mounted supported by dismounted combined arms assault that was to follow. The icons on his common operational picture display indicated the fleeing vehicles had taken an unanticipated route and were going to bypass the planned engagement area. The commander quickly redirected the UAV to reconnoiter a route that his display indicated would allow his 1st platoon to outflank the retreating vehicles while he pursued them with his remaining two platoons.

With the reconnaissance of the UAV assuring the route was clear of obstacles, the 1st platoon advanced rapidly and quickly overtook the fleeing enemy vehicles. Two of the enemy tanks were destroyed with direct fire while the platoon moved parallel to the fleeing enemy force, but the remaining three vehicles found cover behind a low ridge that separated the two forces. Using his embedded planning tools, the 1st platoon leader quickly identified a position in advance of the moving forces that would give him clear shots. Accelerating to speeds of 60 Km/h, the platoon darted in front of the enemy and was there waiting as they crested the ridge and employed direct fire to destroy these enemy forces. With the last of the enemy vehicles confirmed destroyed, the platoon leader ordered the platoon into a traveling over watch formation and continued movement to the west. Figure 79 depicts the mentioned scenario.

Though the remainder of the company was still beyond his direct observation, his Common Operational Picture (COP) display assured him they were moving on parallel routes and that he was well within the supporting range of their fires as

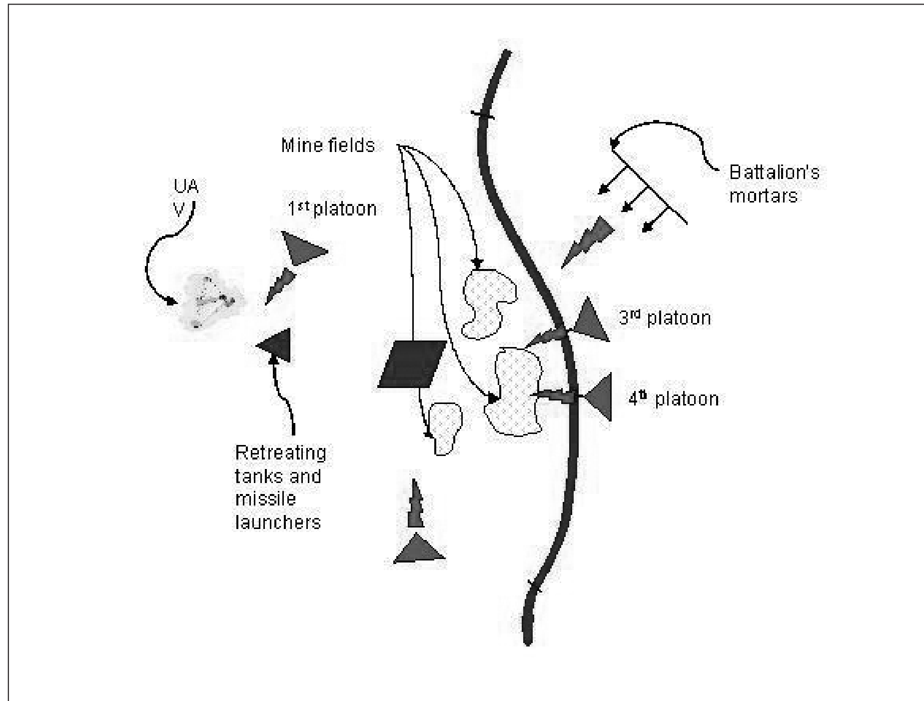


Figure 79: Details of Advances on the Defensive Positions After Mortal Fire

well as those of the battalion's mortars. As they moved towards Agdam, embedded logistics planning tools that had monitored the unit's ammunition usage in the recent engagement automatically transmitted an update to the battalion's logistics center. This constantly updated flow of information enabled the battalion staff to effectively plan en-route re-supply operations that allowed the battalion to maintain its momentum as they continued their pressure on the delaying enemy forces.

In summary, the 1st platoon overtakes and destroys the retreating tanks and missile launchers. The 3rd and 4th platoon force the remaining dismounted enemy forces in the defensive position to flee into the path of the 2nd platoon, ensuring their surrender/destruction. The success in overcoming the defensive position enabled the main elements of the 3rd battalion (Bravo and Echo companies) to overtake the main elements of the ANFRA delaying forces and engage them into a pitched battle, defeating them.

APPENDIX B

NED SOURCE CODE

This appendix contains the source code of the ".ned" files used in this simulation.

B.1 File Generator.ned

```
//---------------------------------------------------------------
// file: generator.ned
//---------------------------------------------------------------
simple Generator
parameters:
  startTime: numeric,
  fromAddr: numeric,   // origin, unique ID within WAN
  totalNodes: numeric; // number of nodes within WAN
// (routers not counted)
gates:
  out: out;
endsimple
```

B.2 File Router.ned

```
//---------------------------------------------------------------
// file: router.ned
//---------------------------------------------------------------
simple Router
parameters:
  startTime: numeric,
  routerID : numeric,
  nodesPerPlane: numeric,
  totalNodes: numeric,
  LANposition : numeric, // Local LAN position
  routerServiceTime: numeric;
gates:
  in: inFromLocal;      // gate #0
  out: outToLocal;      // gate #1
```

```
   in: inFromWirelessPP; // gate #2
   out: outToWirelessPP; // gate #3
   in: inFromWirelessSP; // gate #4
 out: outToWirelessSP;   // gate #5
endsimple
```

B.3 File Satellite.ned

```
 //---------------------------------------------------------------
 // file: satellite.ned
 //---------------------------------------------------------------
 simple Satellite
 parameters:
 startTime: numeric,
 satelliteID : numeric,
 satServiceTime : numeric,
 totalNodes : numeric,
 WGSposition : numeric, // Position at wirelessGS
 WSPposition : numeric; // Position at wirelessSP
 gates:
 in: inBus1; // gate #0 (wirelessGS)
 out: outBus1; // gate #1 (wirelessGS)
 in: inBus2; // gate #2 (wirelessSP)
 out: outBus2; // gate #3 (wirelessSP)
endsimple
```

B.4 File Simplebus.ned

```
 //---------------------------------------------------------------
 // File: simplebus.ned
 // Based on an example by Andras Varga
 //---------------------------------------------------------------
```

```
// Generic bus module. Features:
// - propagation delay modelling (proportional to distance)
// - data rate modelling
// - optional collision modeling
// - optional collision signalling (if turned off, collided
// frames are simply discarded)
// - full duplex or half duplex (simplex) bus. On a full duplex
// bus, frames are assumed to propagate in one direction only
// (upstream or downstream), and transmissions of opposite
// directions don't collide.
// - models several independent channels
//
// Usage:
// Set the parameters of the bus module and connect the stations
// to it. Each station is expected to have a "position" attribute
// which holds the station's distance from one end of the bus.
// There should be NO data rate set for the connecting links!
//
// Frames may have "channel" and "upstream" attributes; if they
// are not present, the default values are 0 and TRUE. "upstream"
// is only significant on a full duplex bus.
//
// The cMessages sent to the bus are interpreted as the start
// of a transmission. Length of transmission is calculated from
// the frame length and the bus data rate.
//
// The cMessages send out by the bus should be interpreted as the
// _end_ of the transmission. Collision signal is an empty cMessage
// with the name "collision". simple SimpleBus
parameters:
busType: string, // Types are: LAN, WPP, WSP, WGS.
  numChannels, // number of independent channels
  wantCollisionModeling, // collision modeling flag
  wantCollisionSignal, // "send collision signals" flag
  isFullDuplex, // channel mode
  delaySecPerMeter, // delay of the bus
  dataRateBps, // data rate of the bus
  gapTime; // minimum gap between consecutive packets.
gates:
  in: in[ ];
  out: out[ ];
endsimple
```

B.5 File Sink.ned

```
//----------------------------------------------------------------
// file: Sink.ned
//----------------------------------------------------------------
simple Sink
gates:
  in: in;
endsimple
```

B.6 File TheNet.ned

```
//----------------------------------------------------------------
// file: theNet.ned
//----------------------------------------------------------------
import "generator.ned";
import "simplebus.ned";
import "sink.ned";
import "router.ned";
import "satellite.ned";
// ------------ Module GroundStation -------------------------
// module GroundStation
parameters:
  nodeID : numeric,
  WGSposition : numeric;
gates:
  out: out;
  in: in;
submodules:
  gen: Generator;
  parameters:
    startTime = ancestor startTime,
    fromAddr = nodeID,
    totalNodes = ancestor nodesPerPlane * ancestor numPlanes;
    display: "i=gen;p=120,49;b=32,30";
    sink: Sink;
    display: "i=sink;p=81,49;b=32,30";
```

```
  connections:
    gen.out --> out;
    sink.in <-- in;
    display: "p=18,2;b=176,102";
endmodule

// ---------- Module computer Node -------------------------
module Node
 parameters:
 nodeID : numeric,
 LANposition : numeric;
 gates:
 out: out;
 in: in;
 submodules:
 gen: Generator;
 parameters:
 startTime = ancestor startTime,
 fromAddr = nodeID,
 totalNodes = ancestor totalNodes;
 display: "i=gen;p=120,49;b=32,30";
 sink: Sink;
 display: "i=sink;p=81,49;b=32,30";
 connections:
 gen.out --> out;
 sink.in <-- in;
 display: "p=18,2;b=176,102";
endmodule

// ------------------ Module Plane -------------------------
 // module Plane
 parameters:
   planeID : numeric,
   nodesPerPlane : numeric,
   totalNodes : numeric,
   WPPposition : numeric,
   WSPposition : numeric,
   routerServiceTime : numeric;
 gates:
   in: inFromWirelessPP;
   out: outToWirelessPP;
   in: inFromWirelessSP;
```

```
  out: outToWirelessSP;
submodules:
  router: Router;
    parameters:
    startTime = ancestor startTime,
    routerID = planeID,
    nodesPerPlane = nodesPerPlane,
    totalNodes = totalNodes,
    LANposition = 10 * nodesPerPlane,
    routerServiceTime = routerServiceTime;
    display: "i=router;p=123,49;b=32,32";
//- - - - - - - - - - - - - - - - - - - - - - - - - -
node: Node[nodesPerPlane];
parameters:
  nodeID = planeID*nodesPerPlane + index,
  LANposition = 10 * index;
  display: "b=38,32;p=43,151,row,45;i=pc";
//- - - - - - - - - - - - - - - - - - - - - - - - - -
ethernetBus: SimpleBus;
parameters:
  busType = "LAN",
  numChannels = 1,
  wantCollisionModeling = 1,
  wantCollisionSignal = 1,
  isFullDuplex = 0,
  delaySecPerMeter = ancestor LANdelay,
  dataRateBps = ancestor LANbandwidth,
  gapTime = ancestor LANgapTime;
gatesizes:
  in[nodesPerPlane + 1],
  out[nodesPerPlane + 1];
  display: "p=88,97;b=156,10,rect";
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
connections:
router.outToLocal --> ethernetBus.in[nodesPerPlane];
router.inFromLocal <-- ethernetBus.out[nodesPerPlane];
router.outToWirelessPP --> outToWirelessPP;
router.inFromWirelessPP <-- inFromWirelessPP;
router.outToWirelessSP --> outToWirelessSP;
router.inFromWirelessSP <-- inFromWirelessSP;
for i=0..nodesPerPlane-1 do
node[i].out --> ethernetBus.in[i];
```

```
 node[i].in <-- ethernetBus.out[i];
 endfor;
 display: "p=2,2;b=168,184";
endmodule

// ----------------- Module TheNet -------------------------
module TheNet
 parameters:
 startTime : numeric,
 nodesPerPlane : numeric,
 numPlanes : numeric,

 LANgapTime : numeric,
 LANbandwidth : numeric,
 LANdelay : numeric,

 WPPgapTime : numeric,
 WPPbandwidth : numeric,
 WPPdelay : numeric,

 WSPgapTime : numeric,
 WSPbandwidth : numeric,
 WSPdelay : numeric,

 WGSgapTime : numeric,
 WGSbandwidth : numeric,
 WGSdelay : numeric,

 satServiceTime : numeric,
 routerServiceTime : numeric;
 submodules:
 //- - - - - - - - - - - - - - - - - - - - - - - - -
 plane: Plane[numPlanes];
 parameters:
   planeID = index,
   nodesPerPlane = nodesPerPlane,
   WPPposition = 100 * index,
   WSPposition = 100 * index,
   totalNodes = nodesPerPlane * numPlanes,
   routerServiceTime = routerServiceTime;
   display: "i=airplane;p=62,90,row,60;b=35,35";
 //- - - - - - - - - - - - - - - - - - - - - - - - -
```

```
  wirelessPP: SimpleBus;
  parameters:
  busType = "WPP",
  numChannels = 1,
  wantCollisionModeling = 1,
  wantCollisionSignal = 1,
  isFullDuplex = 0,
  delaySecPerMeter = WPPdelay,
  dataRateBps = WPPbandwidth,
  gapTime = WPPgapTime;
gatesizes:
  in[numPlanes],
  out[numPlanes];
  display: "p=264,33;b=476,10,rect";
//- - - - - - - - - - - - - - - - - - - - - - - - -
wirelessSP: SimpleBus;
parameters:
  busType = "WSP",
  numChannels = 1,
  wantCollisionModeling = 1,
  wantCollisionSignal = 1,
  isFullDuplex = 0,
  delaySecPerMeter = WSPdelay,
  dataRateBps = WSPbandwidth,
  gapTime = WSPgapTime;
gatesizes:
  in[numPlanes+1],
  out[numPlanes+1];
  display: "p=268,153;b=468,10,rect";
//- - - - - - - - - - - - - - - - - - - - - - - - -
wirelessGS: SimpleBus;
parameters:
  busType = "WGS",
  numChannels = 1,
  wantCollisionModeling = 1,
  wantCollisionSignal = 1,
  isFullDuplex = 0,
  delaySecPerMeter = WGSdelay,
  dataRateBps = WGSbandwidth,
  gapTime = WGSgapTime;
gatesizes:
  in[2],
```

```
    out[2];
    display: "p=272,281;b=476,10,rect";
  //- - - - - - - - - - - - - - - - - - - - - - - - -
  groundStation: GroundStation;
  parameters:
    nodeID = nodesPerPlane * numPlanes,
    WGSposition = 0;
    display: "b=32,32;p=67,215,row,45;i=ground";
  //- - - - - - - - - - - - - - - - - - - - - - - - -
  satellite: Satellite;
  parameters:
    startTime = startTime,
    satelliteID = 1,
    satServiceTime = satServiceTime,
    totalNodes = nodesPerPlane * numPlanes,
    WSPposition = 38300E3, //35800 Km + 2500 Km
    WGSposition = 38300E3; //35800 Km + 2500 Km
    display: "i=satellite;p=315,217;b=32,32";
  //- - - - - - - - - - - - - - - - - - - - - - - - - - -
  connections:
  for i=0..numPlanes-1 do
    plane[i].outToWirelessPP --> wirelessPP.in[i];
    plane[i].inFromWirelessPP <-- wirelessPP.out[i];
    plane[i].outToWirelessSP --> wirelessSP.in[i];
    plane[i].inFromWirelessSP <-- wirelessSP.out[i];
  endfor;
    groundStation.out --> wirelessGS.in[0];
    groundStation.in <-- wirelessGS.out[0];
    satellite.inBus1 <-- wirelessGS.out[1];
    satellite.outBus1 --> wirelessGS.in[1];
    satellite.inBus2 <-- wirelessSP.out[numPlanes];
    satellite.outBus2 --> wirelessSP.in[numPlanes];
    display: "p=10,2;b=508,308";
endmodule

// ----------------------- OTBNet ------------------------
// Instantiates the network
  network OTBNet : TheNet
  parameters:
    startTime = input, // First PDU timestamp in seconds
    nodesPerPlane = input, // Set to 3 in this simulation
    numPlanes = input, // Set to 8 in this simulation
```

```
   LANgapTime = input, // Minimum gap time between frames in the LAN
   LANbandwidth = input, // LAN inside planes (set to 100 Mbps)
   LANdelay = input, // nanosec/meter (set to 70% light speed)

   WPPgapTime = input, // Minimum gap time in the wireless PP
   WPPbandwidth = input, // Wireless bandwidth Plane-to-Plane (PP)
   WPPdelay = input, // nanosec/meter (light speed)

   WSPgapTime = input, // Minimum gap time in the wireless SP
   WSPbandwidth = input, // Wireless bandwidth Satellite-to-Plane(SP)
   WSPdelay = input, // nanosec/meter (light speed)

   WGSgapTime = input, // Minimum gap time in the wireless GS
   WGSbandwidth = input, //Wireless bandwidth Ground-to-Satellite(GS)
   WGSdelay = input, // nanosec/meter (light speed)

   satServiceTime = input,
// Service time per PDU in satellite under best conditions
   routerServiceTime = input;
// Service time per PDU in routers under best conditions
endnetwork
```

B.7 File Omnetpp.ini

```
 [General] network = OTBNet
 ini-warnings = no
 random-seed = 1
 warnings = yes
 snapshot-file = planes.sna
 output-vector-file = planes.vec
 sim-time-limit = 2550s # simulated seconds
 cpu-time-limit = 20h # 20 hours of real cpu time max.
 total-stack-kb = 4096 # 4 MByte, increase if necessary

[Cmdenv]
module-messages = yes
```

```
verbose-simulation = yes
display-update = 0.5s

[Tkenv]
default-run=1
use-mainwindow = yes
print-banners = yes
slowexec-delay = 300ms
update-freq-fast = 10
update-freq-express = 100
breakpoints-enabled = yes

[DisplayStrings]

[Parameters]

[Run 1]
OTBNet.startTime = 1034s
OTBNet.nodesPerPlane = 3
OTBNet.numPlanes = 8

OTBNet.LANgapTime = 50us
OTBNet.LANbandwidth = 100E6 # 100 MBps
OTBNet.LANdelay = 4.761904762ns # nanosec/meter (70% light speed)

OTBNet.WPPgapTime = 50us
OTBNet.WPPbandwidth = 512000
OTBNet.WPPdelay = 3.333333333ns # nanosec/meter (light speed)

OTBNet.WSPgapTime = 50us
OTBNet.WSPbandwidth = 512000
OTBNet.WSPdelay = 3.333333333ns # nanosec/meter (light speed)

OTBNet.WGSgapTime = 50us
OTBNet.WGSbandwidth = 512000
OTBNet.WGSdelay = 3.333333333ns # nanosec/meter (light speed)

OTBNet.satServiceTime = 5us
OTBNet.routerServiceTime = 5us
```

APPENDIX C

AWK SOURCE CODE

This appendix contains the source code of the "`.awk`" files used to parse and extract data from the OTB logger files.

C.1 AWK Script for PDU Parsing

`Awk` program that parses the PDU file generated by OTB and creates files "data*nnnn*.txt" for each generator site identified as *nnnn*.

```
# Process original PDU data files
# with ID numbers added to each "<dis204" (juan.data)

BEGIN {
RS = "\n\\<|\n<";
 # \n is new line, \\< is
 #finally \< and matches the empty string
 #at the beginning of a word.
 origsite = "";
 orighost = "";
 origapplic = "";
 sizeof = "";
 time = "";
 len = "";
 pduName = "";
 pduCount = "?";
 pduId = 0;
 PDUtype = "";
}

/dis204/ {$1 = "<" $1;
 if (orighost == "") orighost = origapplic;
# origin = origsite orighost;
 origin = origsite;
 PDUlength = (len != "" \&\& len != 0 ? len : sizeof);
 if (PDUtype != "") bytes[PDUtype] = bytes[PDUtype] + PDUlength;
 PDUtype = $0;
```

```
 class[PDUtype]++;
 if (origin != "" \&\& time != "" \&\& PDUlength != "" \&\&
    PDUlength != 0)
    {if (!(origin in node)) node[origin] = nodecount++;
     printf "%-12s %5d | %s %5d %s %d\n",
     hextime, PDUlength, time, ++counter[node[origin]],
     pduName, pduId > "data" origin ".txt";
     printf "%-12s %5d | %s %5d %s %d\n",
     hextime, PDUlength, time, counter[node[origin]],
     pduName, pduId > "allpdu.txt";
    }
 else print "dis204 previous to record " NR \
 " has missing parts. pduCount = " pduCount \
 " origin = " origin " time = " time " len = " len;
 origsite = "";
 orighost = "";
 origapplic = "";
 sizeof = "";
 time = "";
 len = "";
 pduName = $0;
 pduCount = "?";
 pduId++;
}

/\.site\>/ {if (origsite == "") origsite = $5}
/\.host\>/ {if (orighost == "") orighost = $5;}
/\.application \>/ {if (origapplic == "") origapplic = $5;}
/\.length\>/ {if (len == "" || len < $5) len = $5}
/\.sizeof\>/ {if (sizeof == "" || sizeof < $5) sizeof = $5}
/\.timestamp\>/ {if (time == "") {hextime = $3; time = $5;}}
/\.pdu_count\>/ {if (pduCount == "?") pduCount = $5;}

END {
 if (orighost == "") orighost = origapplic;
# origin = origsite orighost;
 origin = origsite;
 PDUlength = (len != "" \&\& len != 0 ? len : sizeof);
 bytes[PDUtype] = bytes[PDUtype] + PDUlength;
 for (i in class)
 {printf "%-35s %5d : %8d\n", i, class[i], bytes[i]
         > "pduTypesCount.txt";
```

```
 tot += class[i];
 btot += bytes[i];
}
 printf "\nTotal PDUs = %d, bytes = %d\n", tot, btot
         > "pduTypesCount.txt";

 if (origin != "" \&\& time != "" \&\& PDUlength != "" \&\&
     PDUlength != 0)
     {if (!(origin in node)) node[origin] = nodecount++;
      printf "%-12s %5d | %s %5d %s %d\n",
      hextime, PDUlength, time, ++counter[node[origin]],
      pduName, pduId > "data" origin ".txt";
      printf "%-12s %5d | %s %5d %s %d\n",
      hextime, PDUlength, time, counter[node[origin]],
      pduName, pduId > "allpdu.txt";
     }
 else print "dis204 previous to record " NR \
 " has missing parts. pduCount = " pduCount \
 " origin = " origin " time = " time " len = " len;
 for(origin in node) {
   printf "%5d %s\n", counter[node[origin]], "data" origin ".txt"
          > "nodes.txt";
   close("data" origin ".txt");
 }
 close("nodes.txt");
 close("allpdu.txt");
 system("sort /R nodes.txt /O nodes.txt" );
 system("sort /+21 allpdu.txt /O allpdusort.txt" );
 RS = "\n";
 getline < "nodes.txt";
 system("ren " $2 " data24.txt");
 system("sort /+21 data24.txt /O data24.txt");
 i = 0;
 while ((getline < "nodes.txt") > 0) {
 system("ren " $2 " data" i ".txt");
 system("sort /+21 data" i ".txt" " /O data" i ".txt");
 i+=3;
}
}
```

C.2 AWK Script for Independent Analysis

This `awk` calculates the bandwith required to schedule sets of PDUs at time intervals of at least 2 seconds.

```
 BEGIN {
 tsegment = 2.;  # time interval of 2 seconds
 gap = 0.000050; # 50 microseconds
 tgaps = 0;      # sum of all the gaps in this time interval
 tbytes = 0;     # total of bytes in this time interval
 tcurr = 0;      # current time within the time interval
 PDUcount = 0;   # number of PDUs in this time interval
 firstime = "T"; # flag initially true.
 printf "vector 0 \"band.awk\"
\"Minimum bandwidth requirements over time\" 1\n"
}
 {
 split($4, t, ":"); tsec = t[2]*60+t[3]; # timestamp in seconds
 PDUcount++;
 if (firstime == "T")
 {
 tbytes = $2;
 tcurr = tsec;
 tgaps = gap;
 firstime = "F";
}
 else {
 interval = tsec - tcurr - tgaps; # current size of time interval
 if (interval <= 0 || tsec - tcurr < tsegment)
 {
 tgaps = tgaps + gap;
 tbytes = tbytes + $2;
}
 else
 {
 bw = tbytes*8./interval;
 printf "0 %f %f\n", tcurr, bw
 printf "0 %f %f\n", tsec, bw
 tbytes = $2;
 tcurr = tsec;
```

```
 tgaps = gap;
 PDUcount = 1;
}
}
}
 END {
 if (interval <= 0)
 {tsec += tsegment;
 interval = tsec - tcurr - tgaps;
}
 bw = tbytes*8./interval;
 printf "0 %f %f\n", tcurr, bw;
 printf "0 %f %f\n", tsec, bw;
}
```

C.3 AWK Scripts for Neural Network Processing

This awk script calculates the bandwith required to schedule sets of PDUs at time intervals of at least 2 seconds.

```
* This script reads the file data0.txt and produces data0N45.txt
* containing patterns of 44 consecutive PDU IDs plus a binary
* description of the next PDU ID.

BEGIN {
    PDU["laser"]            = 1;
    PDU["start_resume"]         = 2;
    PDU["stop_freeze"]      = 3;
    PDU["po_task_authorization"]    = 4;
    PDU["po_minefield"]         = 5;
    PDU["fire"]             = 6;
    PDU["detonation"]       = 7;
    PDU["acknowledge"]      = 8;
    PDU["po_delete_objects"]    = 9;
    PDU["minefield"]            = 10;
    PDU["po_message"]       = 11;
```

```
    PDU["signal"]           = 12;
    PDU["aggregate_state"]      = 13;
    PDU["po_simulator_present"]     = 14;
    PDU["po_task_frame"]        = 15;
    PDU["mines"]            = 16;
    PDU["po_point"]             = 17;
    PDU["po_objects_present"]   = 18;
    PDU["po_fire_parameters"]   = 19;
    PDU["iff"]              = 20;
    PDU["po_line"]          = 21;
    PDU["po_parametric_input"]  = 22;
    PDU["po_unit"]          = 23;
    PDU["po_task"]          = 24;
    PDU["transmitter"]      = 25;
    PDU["po_task_state"]        = 26;
    PDU["entity_state"]         = 27;
      count = 0;
      N = 45;
      init = "T";
}

{Hist[count] = PDU[$7];
 count = (count+1)%N;
 if (count == 0) init = "F";
 if (init == "F") {
   for (i=0; i<=N-2; i++)
     printf "%2d ", Hist[(count+i)%N];
   num = Hist[(count+N-1)%N];
   for (i = 0; i <=4; i++) {
     printf "%d ", num%2;
     num = num/2;
   }
   printf "\n";
 }
}


* This script calculates the longest sequence of consecutive PDUs
* bearing the same type
BEGIN {
  init = "T";
  prevType = "";
```

```
  historicType = "";
  historicLength = 0;
  historicPosition = 0;
  seqPos = 1;
  counter = 0;
}

{counter++;
 if (init == "T") {
   prevType = $7;
   currLength = 1;
   init = "F";
 }
 if ($7 == prevType) {
   currLength++;
 }
 else {
   if (currLength > historicLength) {
      historicLength = currLength;
      historicType = prevType;
      historicPosition = seqPos;
   }
   prevType = $7;
   currLength = 1;
   seqPos = counter;
 }
}
```

APPENDIX D

SIMULATOR SOURCE CODE

This appendix contains the source code of the vignette simulator using the OMNeT++ discrete event simulator as the engine, as well as some other auxiliary programs used to prepare the input data and extract specific statistics from the simulator output.

```
//----------------------------------------------------------------------
// file: vecstats.cpp
//----------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#define linesize 100

int main(int argc, char *argv[])
{
 FILE *fd;
 char line[linesize];
 double min, max, sum, avg1, avg2, std, variance, area,
        tInterval, minInterval, maxInterval,
        t1, t2, mt1, mt2, Mt1, Mt2, val1, val2, time1, time2;
 int counter;

 min = DBL_MAX;
 max = 0.;
 sum = area = 0.;
 counter = 0;
 t1 = DBL_MAX;
 t2 = 0.;
 maxInterval = 0.;
 minInterval = DBL_MAX;

 if (argc < 2)
  {
   printf("Usage: %s <band.vec>\n", argv[0]);
   return 1;
  }
```

```
if ((fd = fopen(argv[1], "r")) == NULL)
 {
  printf("Cannot open %s\n", argv[1]);
  return 2;
 }
fgets(line, linesize, fd);
printf("%s", line);

while (fscanf(fd, " %*d %lf %lf", &time1, &val1) != EOF &&
       fscanf(fd, " %*d %lf %lf", &time2, &val2) != EOF )
{
  counter++;
  if (time1 < t1) t1 = time1;
  if (time2 > t2) t2 = time2;\

  if ((time2 - time1) < minInterval)
  {
    minInterval = time2 - time1;
    mt1 = time1;
    mt2 = time2;
  }
  if ((time2 - time1) > maxInterval)
  {
    maxInterval = time2 - time1;
    Mt1 = time1;
    Mt2 = time2;
  }
  if (val1 < min) min = val1;
  if (val1 > max) max = val1;
  sum += val1;
  area += (time2-time1)* (val1+val2)/2.;
}

tInterval = t2 - t1;
avg1 = area / tInterval;
avg2 = sum / counter;
printf("Samples    = %d\n", counter);
printf("Init time  = %11lf\n", t1);
printf("Final time = %11lf\n", t2);
printf("Min time interval = [ %11lf, %11lf ], length = %lf\n",
       mt1, mt2, minInterval);
printf("Max time interval = [ %11lf, %11lf ], length = %lf\n",
```

```
        Mt1, Mt2, maxInterval);
 printf("Minimum bandwidth =%14.1lf\n", min);
 printf("Maximum bandwidth =%14.1lf\n", max);
 printf("Point average     =%14.1lf\n", avg1);
 printf("Area average      =%14.1lf\n", avg2);

 rewind(fd);
 sum = 0.;
 fgets(line, linesize, fd);
 while (fscanf(fd, " %*d %lf %lf", &time1, &val1) != EOF &&
        fscanf(fd, " %*d %lf %lf", &time2, &val2) != EOF )
 {
   sum += (val1 - avg2)* (val1 - avg2);
 }
 variance = sum / (counter - 1.);
 std = sqrt(variance);
 printf("Sample variance   =%14.1lf\n", variance);
 printf("Std deviation     =%14.1lf\n", std);
}


//---------------------------------------------------------------------
// file: pduAnal.c
//---------------------------------------------------------------------

/*This program reads in the original PDU log file as well as the PDU
 summary file corresponding to a given generator, and produces the
 file "extrabyt.txt" that contains pairs of
 (PDU ID, contribution in bytes of that PDU to the group)
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>

#define true 1
#define false 0
#define MAXstring 5000
#define MAXpdus 65000
#define PDUsimilarity 0.49
```

```
int PDUcompareOK (FILE *fd1, FILE *fd2, long int p1, long int p2,
                  int len1, float threshold, int *extraBytes,
                  int *diffLbl);
int readStr(FILE *fd, char a[], char b[], char c[], char d[]);
char buffer[MAXstring*4], buffer2[MAXstring*4];
char pdu1[MAXstring*4], pdu2[MAXstring*4];
char a1[MAXstring], b1[MAXstring], c1[MAXstring], d1[MAXstring];
char a2[MAXstring], b2[MAXstring], c2[MAXstring], d2[MAXstring];

// ----------------------------------------------------------------
int main(int argc, char *argv[]) {
  FILE *fdO1, *fdO2, *fdS, *fdExtraBytes;
  int moreData = 1, pduCount = 0, blockCount, sameBlock, newBlock,
      i, j;
  int ch, endf, len1, len2, id1, id2, extraBytes, diffLbl,
      sumExtraBytes, sumDiffLbl;
  unsigned int timeStamp1, timeStamp2;
  long int fpos[MAXpdus], currpos;
  char  a[MAXstring], b[MAXstring], c[MAXstring], d[MAXstring];
  char type1[50], type2[50];
  double schedTimeSec1, schedTimeSec2, timeSpan,
         hour_equiv = (pow(2.0, 31.0) - 1.0);
  float threshold = PDUsimilarity;
  if (argc != 3) {
     printf("Usage: %s  <file_itsec.data>  <file_dataNN.txt>\n",
             argv[0]);
     return 1;
  }
  fdO1 = fopen(argv[1], "r");  // Original PDU file (the large one)
  fdO2 = fopen(argv[1], "r");  // Original PDU file (the large one)
                               // Same file opened twice
  fpos[0] = -1;
  currpos = ftell(fdO1);
  while ( fgets(buffer, MAXstring*4, fdO1) != NULL ) {
     if (buffer[0] == '<') fpos[++pduCount] = currpos;
     currpos = ftell(fdO1);
  }
//  printf ("Number of PDUs in file %s: pduCount=%d\n",
//           argv[1], pduCount);

// - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  fdS = fopen(argv[2], "r");   // Summary file of PDUs.
```

```
  fscanf(fdS, "%x %d %*[^<]%*s %s %*[^:]: %d",
         &timeStamp2, &len2, type2, &id2);
  schedTimeSec2 = (double)(timeStamp2/2)  * 3600.0 / hour_equiv;
  newBlock = true;
//To store information about extra bytes.
  fdExtraBytes = fopen("extrabyt.txt", "w");

  while (newBlock) {
//newblock = true means not EOF yet and a new empty block is ready.
     len1 = len2;
     id1 = id2;
     strcpy (type1, type2);
     blockCount = 1;
     timeSpan = 0.;
     sumExtraBytes = 0;
     schedTimeSec1 = schedTimeSec2;
     sameBlock = true;
     do {
       endf = fscanf(fdS, "%x %d %*[^<]%*s %s %*[^:]: %d",
                     &timeStamp2, &len2, type2, &id2);
       if (endf == EOF) {newBlock = false; break;}
       schedTimeSec2 = (double)(timeStamp2/2)  * 3600.0 / hour_equiv;
       if ( len1 == len2 && strcmp(type1, type2) == 0 &&
            PDUcompareOK(fdO1, fdO2, fpos[id1], fpos[id2], len1,
                         threshold, &extraBytes, &diffLbl) ) {
          blockCount++;
          sumDiffLbl += diffLbl;
          sumExtraBytes += extraBytes;
          timeSpan = schedTimeSec2 - schedTimeSec1;
          fprintf(fdExtraBytes, "%d, %d\n", id2, extraBytes);
       }
       else sameBlock = false;
     } while (sameBlock);
// if sameblock = false then a new block will start
// printf("PDUId: %5ld %-12s length: %4d #PDUs: %2d extraBytes: \
// %4d diffLabel: %4d timeSpan: %lf\n",
// id1, type1, len1, blockCount, sumExtraBytes, sumDiffLbl,timeSpan);
// printf("%5d, %-20s, %4d, %2d, %4d, %4d, %lf\n",
// id1, type1, len1, blockCount, sumExtraBytes, sumDiffLbl,timeSpan);
  }

  fclose(fdO1);
```

```
  fclose(fdO2);
  fclose(fdS);
  fclose(fdExtraBytes);
  return 0;
}

// ----------------------------------------------------------------------
int PDUcompareOK (FILE *fd1, FILE *fd2, long int p1, long int p2,
    int len1, float threshold, int *extraBytes, int *diffLbl) {
  int diff, diffLabel, eof1, eof2, diffFields;
  float percentSimilar;
    fseek(fd1, p1, SEEK_SET);
    fseek(fd2, p2, SEEK_SET);
    diff = 0;
    diffLabel = 0;
    readStr(fd1, a1, b1, c1, d1);
    readStr(fd2, a2, b2, c2, d2);
    assert(strncmp(a1, "dis204", 6)==0 && strncmp(a2,"dis204",6)==0);
    eof1 = readStr(fd1, a1, b1, c1, d1);
    eof2 = readStr(fd2, a2, b2, c2, d2);
    while (eof1 != EOF && eof2 != EOF &&
           strncmp(a1, "dis204", 6) != 0 &&
           strncmp(a2, "dis204", 6) != 0) {
       diffFields = (strcmp(a1, a2) != 0);
       if (diffFields)
            diffLabel++;
       if (diffFields || strcmp(c1, c2) != 0 || strcmp(d1, d2) != 0 )
            diff += (d1[0]=='\0') ? strlen(c1)/2-1 : strlen(d1)/2-1;
       eof1 = readStr(fd1, a1, b1, c1, d1);
       eof2 = readStr(fd2, a2, b2, c2, d2);
    }
    percentSimilar = (float)(len1 - diff) / (float)len1;
    *extraBytes = diff;
    *diffLbl = diffLabel;
    return (diffLabel == 0 || percentSimilar > threshold);
}

// --------------------------------------------------------
int readStr(FILE *fd, char a[], char b[], char c[], char d[]) {
    a[0] = b[0] = c[0] = d[0] = '\0';
    if (fgets(buffer, MAXstring*4, fd) == NULL) return EOF;
    if (buffer[0] == '\n') return !EOF;
```

```
    if (buffer[0] == '<') { sscanf(buffer, "<%[^>]>", a);
                            return !EOF;
    }
    if (strchr(buffer, '=') == NULL) {
           fgets(buffer2, MAXstring*4, fd);
           strcat(buffer, buffer2);
    }
    if (strchr(buffer,'"') != NULL)
        sscanf(buffer, "%s = \"%[^\"]\" = %s", a, b, c);
    else sscanf(buffer, "%s = %s = %s = %s", a, b, c, d);
    return !EOF;
}

//======================================================================


//----------------------------------------------------------------------
// file: generator.cpp
//----------------------------------------------------------------------

#include <omnetpp.h>
#include <stdio.h>

// Generator simple module class
//
class Generator : public cSimpleModule
{
    // variables used
    FILE *fd, *fdextra, *fdLog;
    char filename[50], msgname[50], pduIniType[50], pduType[50],
         firstCh, c[6], predictedAction;
    char comments[200];
    int commentCount, bundling;
    double dataRateBps, hour_equiv, percentPosSlack;
    simtime_t startTime, gapTime, transmissionTime, schedTimeSec,
              slack, generatorServiceTime, blockWaitTime;
    long pduIniLength, byteFrame_length, bitFrame_length, file_pos;
    unsigned int  eof, num_frames, numNodes, sendTime;
    int pduextra[70000], extralength, pduLenIni, pduLenCurr;
    int frames_sent, frame_counter, pdu_counter, positiveSlack,
        my_address, toAddr, pduID;
    bool firstTime, emptyBlock, blockTimedout, busy;
```

```
    cMessage *readyToSend, *blockTimeout, *msg1;
    cOutVector slackTime;

    // member functions
    Module_Class_Members(Generator,cSimpleModule,0)
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();

private:
    void PDUrecord1();
    void PDUrecord2();
};


Define_Module( Generator );

//=====================================================================
void Generator::initialize()
{
    commentCount = 0;
    for (pduID=0; pduID<70000; pduID++)
        pduextra[pduID] = 0;
    fdextra = fopen("juanTgz\\juanTgz3\\extrab.txt", "r");
    fdLog = fopen("juanTgz\\juanTgz4\\PDUlog.txt", "w");
    int counterextra = 0;
    while (fscanf(fdextra, "%d, %d", &pduID, &extralength) != EOF) {
          pduextra[pduID] = extralength;
          counterextra++;
    }
    startTime = par("startTime");
    blockWaitTime = par("blockWaitTime");
    generatorServiceTime = par("generatorServiceTime");
    gapTime = gate("out")->toGate()->toGate()->ownerModule()
              ->par("gapTime");
    my_address = par("fromAddr");
    c[4] = 'W'; c[5] = 'S';
    bundling = par("bundling"); printf("bundling = %d\n", bundling);
    if (bundling < 1 || bundling > 6 )
        {printf("Error in generator %d, bundling = %d\n",
                 my_address, bundling);
         return;}
```

```
    numNodes = par("totalNodes");
    dataRateBps = (double)gate("out")->
                  toGate()->toGate()->ownerModule()
                  ->par("dataRateBps");
    printf("Generator my_address=%d, numNodes=%d startTime=%lf "
          "blockWaitTime=%lf generatorServiceTime=%lf gapTime=%lf\n",
           my_address, numNodes, startTime, blockWaitTime,
           generatorServiceTime, gapTime);

    hour_equiv = (pow(2.0, 31.0) - 1.0);
    frames_sent = 0;
    frame_counter = 0;
    pdu_counter = 0;
    positiveSlack = 0;
    toAddr = -1;            // all packets are broadcasted

    slackTime.setName("Slack Time to Send Next Message");
    firstTime = true;
    emptyBlock = true;
    blockTimedout = false;
    busy = false;
    readyToSend =   new cMessage("readyToSend");
    blockTimeout =  new cMessage("blockTimeout");

    sprintf(filename, "juanTgz\\juanTgz4\\dataNew%d.txt",my_address);
    if ((fd = fopen(filename, "r")) != NULL)
    {
        scheduleAt (startTime, readyToSend);  // schedule first event
//      printf("Generator my_address=%d scheduled readyToSend.\
 Initialization completed.\n", my_address);
    }
}


//====================================================================
void Generator::handleMessage(cMessage *msg)
{
    if(msg == blockTimeout)
    {
        if (busy) {
           blockTimedout = true; //block is sent at next readyToSend
           return;
```

```
        }  // end of busy status
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        // status is idle (not busy)
        msg1->setTimestamp();    // block will be sent immediately
        transmissionTime = (double)msg1->length() / dataRateBps;
        send(msg1,"out");
        frames_sent++;
        PDUrecord2();
        emptyBlock = true;
        blockTimedout = false;  // block was just sent
        busy =  true;
        if (eof != EOF) {    // if EOF and block not empty,
                             // then readyToSend was not scheduled
           cancelEvent(readyToSend);  //remove previous (future time)
                                      // readyToSend
        }
        scheduleAt(simTime() + transmissionTime + gapTime +
                  generatorServiceTime, readyToSend);
        return;
    }  //end of msg == blockTimeout
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// msg is not blockTimeout, should be readyToSend
    if (msg == readyToSend)
    {
        if (blockTimedout) {
           //current block has priority over new PDUs
           msg1->setTimestamp();   // block will be sent immediately
           transmissionTime = (double)msg1->length() / dataRateBps;
           send(msg1,"out");
           frames_sent++;
           PDUrecord2();
           emptyBlock = true;
           blockTimedout = false;  // block was just sent
           busy =  true;
           scheduleAt(simTime() + transmissionTime + gapTime +
                     generatorServiceTime, readyToSend);
           return;
        }  //end of blockTimedout
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// msg is readyToSend & not blockTimedout
```

```
        firstCh = getc(fd);     //skips over comments indicated by
                                // '%' in first char
        while (firstCh == '#') {
            fgets(comments, 200, fd);
            commentCount++;
            printf("Comments # %d in %s are: %s\n",
                    commentCount, filename, comments);
            firstCh = getc(fd);
        }
        ungetc(firstCh, fd);
        file_pos = ftell(fd);
        eof = fscanf(fd,
              "%x %ld | %*s %*d <dis204 %s PDU>: %d %c %c %c %c",
              &sendTime, &byteFrame_length, pduType, &pduID,
              &c[0],&c[1],&c[2],&c[3]);
            // c[0] is neural network (column 1)
            // c[1] is type (column 2)
            // c[2] is type and length (column 3)
            // c[3] is type, length and timestamp (column 4)
            // c[4] = W Always-Wait
            // c[5] = S Always-Send
        predictedAction = c[bundling-1];
        if (eof == EOF)  {
           if (emptyBlock) {       // end of generator simulation
              percentPosSlack = (double)positiveSlack *
                                100.0 / (double)pdu_counter;
              printf("EOF in file %11s at time %lf, positive slack "
              "frames=%6d(%5.2lf%%), total PDUs=%6d, total frames"
              " built=%6d, total frames sent=%6d\n",
              filename, simTime(), positiveSlack, percentPosSlack,
              pdu_counter, frame_counter, frames_sent);
              fclose(fdLog);
           }  // end of emptyBlock
           else {        // block is not empty
              busy = false;
              cancelEvent(blockTimeout);
              scheduleAt(simTime(), blockTimeout);
           }
           return;
        }  // end eof == EOF
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
//      msg = readyToSend  & not blockTimedout & not EOF
//      we read a new PDU from the summary file.
       //conversion from OTB units to seconds
        schedTimeSec = (double)(sendTime/2) * 3600.0 / hour_equiv;
        bitFrame_length = byteFrame_length * 8;
        slack = schedTimeSec - simTime();
        if (firstTime)  {           // first time this particular PDU
                                    // was read from the input file.
            pdu_counter++;
            slackTime.record(slack);
            if (slack >= 0) positiveSlack++;
            firstTime = false;   // this particular PDU
                                 //won't be recorded again.
        }
        if (slack > 0.)  {
           busy = false;   // we will be idle for a while
           if (fseek(fd, file_pos, SEEK_SET)) perror("Fseek failed");
           // next packet is scheduled at timestamp in PDU
           scheduleAt(schedTimeSec, readyToSend);
           return;
        }  // end of slack > 0.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

//  msg = readyToSend & not blockTimedout & not EOF & slack <= 0.
// This PDU must be grouped for replication
        firstTime = true; // to record slack for next PDU.
        if (emptyBlock) {
           //This PDU will be the first in the new block
          sprintf(msgname,"Data%d F%d T%d",
                  ++frame_counter, my_address, toAddr);
          msg1 = new cMessage(msgname);
          msg1->setLength(bitFrame_length);
          PDUrecord1();
          strcpy(pduIniType,pduType);
          pduIniLength = byteFrame_length;
          if (predictedAction == 's' || predictedAction == 'S')
          {  // predicted action = send
             msg1->setTimestamp();
             transmissionTime = (double)msg1->length() / dataRateBps;
             send(msg1,"out");
             frames_sent++;
             PDUrecord2();
```

```
            busy = true;
            scheduleAt(simTime() + transmissionTime + gapTime +
                       generatorServiceTime, readyToSend);
         }
         else {                // predicted action = wait
            scheduleAt(simTime() + blockWaitTime, blockTimeout);
            busy = false;
            emptyBlock = false;
            scheduleAt(simTime()+generatorServiceTime, readyToSend);
         }
         return;
       }  // end of emptyBlock
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// msg = readyToSend & not blockTimedout &
// not EOF & slack <= 0. & not emptyBlock
       if ((strcmp(pduIniType,pduType)==0) &&
          (pduIniLength==byteFrame_length)) { //compatible PDU
//grouping (bundling)
          msg1->setLength(msg1->length()+(pduextra[pduID]*8));
          PDUrecord1();
          if (predictedAction == 's' || predictedAction == 'S') {
             msg1->setTimestamp();   // predicted action = send
             transmissionTime= (double)msg1->length() / dataRateBps;
             send(msg1,"out");
             frames_sent++;
             PDUrecord2();
             cancelEvent(blockTimeout);
             emptyBlock = true;
             busy = true;
             scheduleAt(simTime() + transmissionTime + gapTime +
                        generatorServiceTime, readyToSend);
          }         // end of predictedAction = send
          else {    // predicted action = wait
             busy = true;
             scheduleAt(simTime()+generatorServiceTime,readyToSend);
          }         // end of predicted action = wait
          return;
       } //  end of compatible PDU
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// msg = readyToSend & not blockTimedout &
```

```
// not EOF & slack <= 0. & not emptyBlock & PDU not compatible
        msg1->setTimestamp();
        transmissionTime = (double)msg1->length() / dataRateBps;
        send(msg1,"out");
        frames_sent++;
        PDUrecord2();
        cancelEvent(blockTimeout);
        sprintf(msgname,"Data%d F%d T%d",
        ++frame_counter, my_address, toAddr);
        msg1 = new cMessage(msgname);
        msg1->setLength(bitFrame_length);
        PDUrecord1();
        strcpy(pduIniType,pduType);
        pduIniLength = byteFrame_length;
        if (predictedAction == 's' || predictedAction == 'S')
        {  // predicted action = send
           // this 1-PDU block is considered to have timedout
           blockTimedout = true;
        }
        else {        // predicted action = wait
           scheduleAt(simTime() + blockWaitTime +
                      generatorServiceTime, blockTimeout);
        }             // end of predicted action = wait
        busy = true;
        scheduleAt(simTime() + transmissionTime + gapTime +
                   generatorServiceTime, readyToSend);
        return;  //  msg = readyToSend & not EOF & slack <= 0.
    } // end msg == readyToSend
//- - - - - - - - - - - - - - - - - - - - - - - - - - -  - - -

    printf("Generator %d: Unrecognized message\n", my_address);
    return;
}

//=====================================================================
void Generator::finish()  {
  ev << "Generator " << my_address << ": No of frames sent = "
     << frame_counter << endl;
}

//=====================================================================
void Generator::PDUrecord1()  {
```

```
  if (my_address == 24) fprintf(fdLog, "%d ", pduID);
}

//======================================================================
void Generator::PDUrecord2()  {
  double t, seconds;
  int minutes;
  t = simTime();
  minutes = (int) t/60.;
  seconds = t - minutes*60.;

  if (my_address == 24) {
      double t, seconds;
      int minutes;
      t = simTime();
      minutes = (int) (t/60.);
      seconds = t - minutes*60.;
      fprintf(fdLog, ": %lf (:%d:%.3lf) | \tframes:%d\n",
              t, minutes, seconds, frames_sent);
  }
}

//----------------------------------------------------------------
// file: sink.cpp
//----------------------------------------------------------------

#include <omnetpp.h>


//
// Sink simple module class
//
class Sink : public cSimpleModule
{
    int my_address, from, to, collisionCounter, framesReceived,
        unrecognized, wrongAddress;
    double travelTime;
    cMessage *collision;
    char *p;

    // member functions
    Module_Class_Members(Sink,cSimpleModule,0)
```

```
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();

    cDoubleHistogram *travelDist;
    cOutVector travelHist;
    cStdDev travelStats;

    cOutVector collHistAccum;

};

Define_Module( Sink );

void Sink::initialize()
{
    collisionCounter = 0;
    my_address = parentModule()->par("nodeID");
    collision = new cMessage("collision");
    travelDist = new cDoubleHistogram(
                 "Travel Time Distribution at destination", 100);
    travelDist->setRange(0, 100);
    travelHist.setName("Travel Time History");
    travelStats.setName("travel Stats");
    collHistAccum.setName("Collision Accumulation");
    framesReceived = 0;
    unrecognized = 0;
    wrongAddress = 0;
}

void Sink::handleMessage(cMessage *msg)
{
    // msg == collision
    if (strcmp(msg->name(), collision->name()) == 0)
    {
        collisionCounter++;
        collHistAccum.record (collisionCounter);
        delete msg;
        return;
    }

    p = strchr(msg->name(),'F');
```

```
    if (p == NULL)
    {   ev<<"Sink["<<my_address<<"] unrecognized deleted "
          <<msg->name()<<endl;
        delete msg;     // unrecognized message, considered an error
        unrecognized++;
    }
    else                    // p != NULL, this is a regular message
    sscanf(p, "F%d T%d", &from, &to);

    if (to == -1 || to == my_address)
    {
//      ev << "Sink[" << my_address << "] Frame " << msg->name()
            <<" at T = " << simTime() << endl;
        travelTime = simTime() - msg->timestamp();
        // travel time travelStatsistics collection
        travelDist->collect (travelTime);
        travelHist.record(travelTime);
        travelStats.collect(travelTime);
        framesReceived++;
    }
    else wrongAddress++;

    delete msg;
}

void Sink::finish()
{
    long num_samples;
    double smallest, largest, mean,
           standard_deviation, variance;

    ev << endl << endl<< "*** Module: " << fullPath()<<"***" << endl;
    ev << "Total arrivals: " << travelDist->samples() << endl;
    ev << "Estimation of the travel stationary \
    distribution of travel time.\n";
    ev << "Travel time, # of messages, estimated \
    probability density function.\n";
    for(int i=0; i<travelDist->cells(); ++i)
    {   if(travelDist->cell(i) > 0)
        {   ev << i << ":\t" << travelDist->cell(i);
            ev << "\t" << travelDist->cellPDF(i) << endl;
```

```
        }
    }
    recordStats("Travel Time Distribution Statistics", travelDist);
    ev << "Travel Time Statistics" << endl;
    num_samples = travelStats.samples();
    smallest = travelStats.min();
    largest = travelStats.max();
    mean = travelStats.mean();
    standard_deviation = travelStats.stddev(),
    variance = travelStats.variance();
    ev << "Number of samples: " << num_samples          << endl;
    ev << "Smallest time: "     << smallest             << endl;
    ev << "Largest time: "      << largest              << endl;
    ev << "Mean value: "        << mean                 << endl;
    ev << "Standard Dev: "      << standard_deviation   << endl;
    ev << "Variance: "          << variance             << endl;
    printf("Sink %d: Total frames received=%d, total collisions \
    detected=%d total unrecognized=%d wrong address=%d\n",
           my_address, framesReceived, collisionCounter,
           unrecognized, wrongAddress);
}

//---------------------------------------------------------------
// file: router.cpp
//---------------------------------------------------------------

#include <omnetpp.h>
#include <string.h>

//
// Router simple module class
//
class Router : public cSimpleModule
{
    int routerID, nodesPerPlane, totalNodes;
    int inf, sup;
    int from, to, inGate, outGate;
    // 3 communication channels.
    int collisionCount[3], collisionCountNonReset[3];
    double  startTime, routerServiceTime, transmissionTime,
            collInterval, gapTime[4], dataRate[4];
    int fromLan, toLan, fromSP, toSP, fromPP, toPP;  //frame counters
```

```
    cQueue queue;
    cMessage *sendNow, *readyToSend, *collision,
             *collStatsNow, *msg1, *msg2;

    cDoubleHistogram *jobDist;
    cOutVector jobsInSys;
    cStdDev stat;

    cDoubleHistogram *collDist[3];
    cOutVector collInSys[3];

    cOutVector collHistAccum;

    // member functions
    Module_Class_Members (Router, cSimpleModule,0)
    virtual void initialize ();
    virtual void finish ();
    virtual void handleMessage (cMessage *msg);
    void serveMessage();
    int outputGate (int inGate, int from, int to);
};

Define_Module( Router );



//===============================================================
void Router::initialize()
{
    int i;
    startTime = par("startTime");
    routerID        = par("routerID");
    nodesPerPlane   = par("nodesPerPlane");
    totalNodes      = par("totalNodes");
    routerServiceTime = par("routerServiceTime");

    gapTime[0]  = gate("outToLocal")->toGate()->ownerModule()
                  ->par("gapTime");
    gapTime[1]  = gate("outToWirelessPP")->toGate()->toGate()
                  ->ownerModule()->par("gapTime");
    gapTime[2]  = gate("outToWirelessSP")->toGate()->toGate()
```

```
              ->ownerModule()->par("gapTime");
gapTime[3]  = gapTime[1] > gapTime[2] ? gapTime[1] : gapTime[2];

dataRate[0] = (double)gate("outToLocal")     ->toGate()
              ->ownerModule()->par("dataRateBps");
dataRate[1] = (double)gate("outToWirelessPP")->toGate()->toGate()
              ->ownerModule()->par("dataRateBps");
dataRate[2] = (double)gate("outToWirelessSP")->toGate()->toGate()
              ->ownerModule()->par("dataRateBps");
dataRate[3]= dataRate[1]<dataRate[2] ? dataRate[1] : dataRate[2];

collision    = new cMessage("collision");
collStatsNow = new cMessage("collStatsNow");
readyToSend  = new cMessage("readyToSend");
sendNow      = new cMessage("sendNow");

inf = nodesPerPlane * routerID;
sup = inf + nodesPerPlane - 1;
// msg1 = NULL because initial state is "readyToSend"
msg1 = NULL;

jobDist = new cDoubleHistogram(
          "Queue Message Distribution (router)", 100);
jobDist->setRange(0, 100);
jobsInSys.setName("Messages in System (router)");
stat.setName("stat");

{   char *titles[3] =   { "Collisions at inFromLocal (Ethernet)",
                          "Collisions at inFromWirelessPP",
                          "Collisions at inFromWirelessSP" };
    for (i = 0; i<3; i++)
    {   collisionCount[i] = 0;
        collisionCountNonReset[i] = 0;
        collDist[i] = new cDoubleHistogram(titles[i], 100);
        collDist[i]->setRange(0, 100);
        collInSys[i].setName(titles[i]);
    }
}
collHistAccum.setName("Collision Accumulation");

// count collisions in 1-second intervals
collInterval = 1.;
```

```
    // frame counters set to 0
    fromLan = toLan = fromSP = toSP = fromPP = toPP = 0;
    // first event to request collision statistics.
    scheduleAt(collInterval+startTime, collStatsNow);
}


//=================================================================
void Router::handleMessage(cMessage *msg)
{
    if (strcmp(msg->name(),collision->name())==0) // msg == collision
    {   inGate = msg->arrivalGate()->id() /2; // inGate = 0 or 1 or 2
        collisionCount[inGate]++;
        collisionCountNonReset[inGate]++;
        collHistAccum.record (collisionCountNonReset[0] +
                              collisionCountNonReset[1] +
                              collisionCountNonReset[2]);
        delete msg;
        return;
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Statistics collection requested now
    else if (msg == collStatsNow)
    {   for (int i=0; i<3; i++)
        {   collDist[i]->collect (collisionCount[i]);
            collInSys[i].record(collisionCount[i]);
    // starts a new count for the next interval
            collisionCount[i] = 0;
        }
        scheduleAt(simTime()+collInterval, collStatsNow);
        return;
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else if (msg == sendNow)
    {
        switch (outGate)
        {
            case 0:
                send(msg1, "outToLocal");
                toLan++;
```

```
                break;

            case 1:
                send(msg1, "outToWirelessPP");
                toPP++;
                break;

            case 2:
                send(msg1, "outToWirelessSP");
                toSP++;
                break;

            case 3:
                msg2 = (cMessage *) msg1->dup();
                send(msg1, "outToWirelessPP");
                toPP++;
                send(msg2, "outToWirelessSP");
                toSP++;
                break;
        }
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else if (msg == readyToSend) // last gapTime has elapsed
    {
        if(queue.empty()) //There are no remaining messages in queue
        {
            msg1 = NULL;
        }

        else
        {
            msg1 = (cMessage *) queue.pop();
// schedules a sendNow and readyToSend for msg1
            serveMessage();
        }
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else       // msg == regular message || unrecognized
    {
// to ignore messages sent to satellite from other planes
```

```
      char *p = strchr(msg->name(),'F');
      sscanf(p, "F%d T%d", &from, &to);
      inGate = msg->arrivalGate()->id();
// gate #4: inFromWirelessSP
      if ((inGate == 4) && (from != totalNodes))
      {
          delete msg;
          return;
      }

// msg arrived while server is idle, current state is "readyToSend"
      if (msg1 == NULL)
// Statistics collection: queue length was 0
      {   jobDist->collect(0);
          jobsInSys.record(0);
          stat.collect(0.);
          msg1 = msg;     //msg will be serviced immediately
          serveMessage();
          //schedules a sendNow and readyToSend for msg1
      }

      else   // Arrival while server is busy
      {
// n msgs in queue + 1 being serviced
          jobDist->collect(queue.length()+1);
          jobsInSys.record(queue.length()+1);
          stat.collect(queue.length()+1.);
          queue.insert( msg );
      }
    }   // end of regular message

}       // end handleMessage


//=============================================================
void Router::serveMessage()
{
    char *p = strchr(msg1->name(),'F');
    if (p == NULL)      // unrecognized message, considered an error
    {
        ev<<"Router["<<routerID<<"] unrecognized deleted "
          <<msg1->name()<<endl;
```

```
        delete msg1;
        scheduleAt( simTime(), readyToSend );
        return;
    }

    sscanf(p, "F%d T%d", &from, &to);
// inGate: 0/2 = 0, 2/2 = 1 or 4/2 = 2
    inGate = (msg1->arrivalGate()->id()) / 2;
    if (inGate==0)fromLan++;
    if (inGate==1)fromPP++;
    if (inGate==2)fromSP++;
// outGate = -1, 0, 1, 2, 3
    outGate = outputGate(inGate, from, to);
    if (outGate < 0)
    {
        delete msg1;
        scheduleAt( simTime(), readyToSend );
        return;
    }

    transmissionTime = msg1->length() / dataRate[outGate];
    scheduleAt( simTime() + routerServiceTime, sendNow );
    scheduleAt( simTime() + routerServiceTime + transmissionTime
              + gapTime[outGate], readyToSend );

}


//==========================================================
int Router::outputGate(int inGate, int from, int to)
{
    switch (inGate)
    {
        case 0:                                    // inFromLocal
            if (to == -1)            return 3;   // outToWirelessSP
                                                // and outToWirelessPP
            if (to == totalNodes)    return 2;   // outToWirelessSP
            if (to < inf || to > sup) return 1;   // outToWirelessPP
            return -1;                            // delete message
            break;

        case 1:                                  // inFromWirelessPP
```

```
            if (to == -1)               return 0;   // outToLocal
// delete, this case shoud not occur
            if (to == totalNodes)       return -1;
            if (inf <= to && to <= sup) return 0;   // outToLocal
            return -1;                             // delete message
            break;

        case 2:                                   // inFromWirelessSP
            if (
                (from == totalNodes) &&
                ((to == -1) || (inf <= to && to <= sup))
// from satellite (groundStation) to local broadcast
               ) return 0;
            return -1;                             // delete message
            break;
    }
    return -2;      // unreachable code to eliminate C++ warning.
}


//=========================================================
void Router::finish()
{
    long num_samples;
    double smallest, largest, mean, standard_deviation, variance;

    ev << endl << endl<< "*** Module: " << fullPath()<<"***" << endl;
    ev << "Total arrivals:\t" << jobDist->samples() << endl;
    ev << "Total collisions detected:"<<endl;
    ev << "At inFromLocal: " << collisionCountNonReset[0]<<endl;
    ev << "At wirelessPP: "  << collisionCountNonReset[1]<<endl;
    ev << "At wirelessSP: " << collisionCountNonReset[2]<<endl<<endl;

    ev << "Estimation of the stationary  distribution of messages \
    as observed by an arrival.\n";
    ev << "Queue length, # arrivals that saw n messages in queue, \
    estimated probability density function.\n";
    for(int i=0; i<jobDist->cells(); ++i)
    {   if(jobDist->cell(i) > 0)
        {   ev << i << ":\t" << jobDist->cell(i);
            ev << "\t" << jobDist->cellPDF(i) << endl;
        }
```

```
    }
    recordStats("Message Distribution Statistics", jobDist);
    ev << "Queue length statistics" << endl;
    num_samples = stat.samples();
    smallest = stat.min();
    largest = stat.max();
    mean = stat.mean();
    standard_deviation = stat.stddev(),
    variance = stat.variance();
    ev << "Number of samples: " << num_samples          << endl;
    ev << "Smallest queue: "    << smallest             << endl;
    ev << "Largest queue: "     << largest              << endl;
    ev << "Mean value: "        << mean                 << endl;
    ev << "Standar Dev: "       << standard_deviation   << endl;
    ev << "Variance: "          << variance             << endl;
    printf("Router %d: frames fromLan=%d, toLan=%d, fromSP=%d, \
    toSP=%d, fromPP=%d, toPP=%d, in queue=%d\n",
    routerID, fromLan, toLan, fromSP, toSP, fromPP,
    toPP, queue.length());
}

//---------------------------------------------------------------
// file: satellite.cpp
//---------------------------------------------------------------

#include <omnetpp.h>
#include <string.h>

//
// Satellite simple module class
//
class Satellite : public cSimpleModule
{
    //arrays are of length 2 because of the 2 communication channels.
    int satelliteID, totalNodes;
    double startTime, satServiceTime, transmissionTime,
           gapTime[2], dataRate[2];
    double WSPposition, WGSposition, collInterval;
    int from, to, inGate, outGate, numGate;
    int collisionCount[2], collisionCountNonReset[2], byteCount,
        framesToGS, framesToPlanes, framesReceivedFromGS,
        framesReceivedFromSP, unrecognized;
```

```
    char *p;

    cQueue queue;
    cMessage *sendNow, *readyToSend, *collision, *collStatsNow,*msg1;
    cDoubleHistogram *jobDist, *byteDist;
    cOutVector jobsInSys, bytesInSys;
    cStdDev msgStat, byteStat;
    cDoubleHistogram *collDist[2];
    cOutVector collInSys[2];

    // member functions
    Module_Class_Members(Satellite, cSimpleModule,0)
    virtual void initialize();
    virtual void finish();
    virtual void handleMessage(cMessage *msg);
    void serveMessage();
    int outputGate (int inGate, int from, int to);


};

Define_Module( Satellite );

//=================================================================
void Satellite::initialize()
{
    int i;
    startTime = par("startTime");
    satServiceTime = par("satServiceTime");
    sendNow = new cMessage("sendNow");
    collision = new cMessage("collision");
    collStatsNow = new cMessage("collStatsNow");
    readyToSend  = new cMessage("readyToSend");

    gapTime[0]  = gate("outBus1")->toGate()->ownerModule()
                  ->par("gapTime");
    gapTime[1]  = gate("outBus2")->toGate()->ownerModule()
                  ->par("gapTime");

    dataRate[0] = (double)gate("outBus1")->toGate()->ownerModule()
                  ->par("dataRateBps");
    dataRate[1] = (double)gate("outBus2")->toGate()->ownerModule()
```

```
                ->par("dataRateBps");

    satelliteID = par("satelliteID");
// totalNodes = 3*8 = 24, but 0,...,24 = 25 nodes
    totalNodes = par("totalNodes");
//  WSPposition  = par("WSPposition");
//  WGSposition  = par("WGSposition");
    msg1 = NULL;

    jobDist = new cDoubleHistogram(
              "Queue Message Distribution (satellite)", 100);
    jobDist->setRange(0, 100);
    jobsInSys.setName("Messages in System (satellite)");
    byteDist = new cDoubleHistogram(
              "Queue Byte Distribution (satellite)", 100);
    byteDist->setRange(0, 100);
    bytesInSys.setName("Bytes in System (satellite)");

    {   char *titles[2] =   { "Collisions at wirelessGS",
                              "Collisions at wirelessSP" };

        for (i = 0; i<2; i++)
        {   collisionCount[i] = 0;
            collisionCountNonReset[i] = 0;
            collDist[i] = new cDoubleHistogram(titles[i], 100);
            collDist[i]->setRange(0, 100);
            collInSys[i].setName(titles[i]);
        }
    }
    framesToGS = 0;      // to count frames sent to Ground Station
    framesToPlanes = 0;
    framesReceivedFromGS = 0;
    framesReceivedFromSP = 0;
    unrecognized = 0;
    byteCount = 0;                      // counts bytes in queue.
    collInterval = 1.;     // count collisions in 1-second intervals
// first event to request collision statistics.
    scheduleAt(collInterval+startTime, collStatsNow);

}

//================================================================
```

```
void Satellite::handleMessage(cMessage *msg)
{
    if (strcmp(msg->name(), collision->name())==0) //msg == collision
    {
        inGate = msg->arrivalGate()->id() /2;       //inGate = 0 or 1
        collisionCount[inGate]++;
        collisionCountNonReset[inGate]++;
        delete msg;
        return;
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else if (msg == collStatsNow)
    //Statistics collection requested now
    {   for (int i=0; i<2; i++)
        {   collDist[i]->collect (collisionCount[i]);
            collInSys[i].record(collisionCount[i]);
// starts a new count for the next interval
            collisionCount[i] = 0;
        }
        scheduleAt(simTime()+collInterval, collStatsNow);
        return;
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else if (msg == sendNow)
    {
        switch (outGate)
        {
            case 0:
                send(msg1, "outBus1");  // wirelessGS
                framesToGS++;
                break;

            case 1:
                send(msg1, "outBus2");  // wirelessSP
                framesToPlanes++;
                break;
        }
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
    else if (msg == readyToSend)    // last gapTime has elapsed
    {
        if ( queue.empty() )
        // There are no remaining messages in queue
        {
            msg1 = NULL;
            if (byteCount != 0)
          printf("Satellite: Error, empty queue has byteCount=%d\n",
                    byteCount);
        }

        else
        {
            msg1 = (cMessage *) queue.pop();
// subtracts # bytes taken from the queue
            byteCount -= msg1->length()/8;
// schedules a sendNow and readyToSend for msg1
            serveMessage();
        }
    }

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    else            // msg == regular message or unrecognized
    {
// msg arrived while server is idle, current state is "readyToSend"
      if (msg1 == NULL)
      {
// Statistics collection: queue length was 0
          jobDist->collect(0);
          jobsInSys.record(0);
          msgStat.collect(0.);
// Statistics collection: queue length was 0
          byteDist->collect(0);
          bytesInSys.record(0);
          byteStat.collect(0.);
          msg1 = msg;           // msg will be serviced immediately
// schedules a sendNow and readyToSend for msg1
          serveMessage();
      }
      else                  // Arrival while server is busy
      {
// n msgs in queue + 1 being serviced
```

```
          jobDist->collect(queue.length()+1);
          jobsInSys.record(queue.length()+1);
          msgStat.collect(queue.length()+1.);
// accumulates # bytes in new message
          byteCount += msg->length()/8;
// n msgs in queue + 1 being serviced
          byteDist->collect(byteCount);
          bytesInSys.record(byteCount);
          byteStat.collect(byteCount);
          queue.insert( msg );
      }
    }   // end of regular message

}       // end handleMessage


//==============================================================
void Satellite::serveMessage()
{
    char *p = strchr(msg1->name(),'F');
    if (p == NULL)    // unrecognized message, considered an error
    {
        ev<<"Satellite: unrecognized message deleted "<<endl;
        delete msg1;
        unrecognized++;
        scheduleAt( simTime(), readyToSend );
        return;
    }

    sscanf(p, "F%d T%d", &from, &to);
    inGate = (msg1->arrivalGate()->id()) / 2; //inGate: 0/2=0, 2/2=1
    if (inGate==0) framesReceivedFromGS++;
    else framesReceivedFromSP++;
    outGate = outputGate(inGate, from, to);   // outGate = -1, 0, 1
    if (outGate < 0)
    {
        delete msg1;
        unrecognized++;
        scheduleAt( simTime(), readyToSend );
        return;
    }
```

```
    transmissionTime = msg1->length() / dataRate[outGate];
    scheduleAt( simTime() + satServiceTime, sendNow );
    scheduleAt( simTime() + satServiceTime + transmissionTime +
                gapTime[outGate], readyToSend );

}


//==============================================================
int Satellite::outputGate(int inGate, int from, int to)
{
    switch (inGate)
    {
        case 0:                          // inBus1 (wirelessGS)
            if (to < totalNodes) return 1;  // WirelessSP
            return -1;                      // delete message
            break;

        case 1:                               // inBus2 (wirelessSP)
            if (to == -1 || to == totalNodes)
                 return 0; // inBus1 wirelessGS
            return -1;                           // delete message
            break;
    }

    return -2;      // unreachable code to eliminate C++ warning.
}


//=========================================================
void Satellite::finish()
{
    long num_samples;
    double smallest, largest, mean, standard_deviation, variance;

    ev << endl << endl<< "*** Module: " << fullPath()
       << "***" << endl;
    ev << "Total arrivals:\t" << jobDist->samples() << endl;
    ev << "Total collisions detected:"<<endl;
    ev << "At wirelessGS: " << collisionCountNonReset[0]
       <<endl;
    ev << "At wirelessSP: " << collisionCountNonReset[1]
```

```
    <<endl<<endl;
  ev << "Estimation of the stationary  distribution of \
  messages as observed by an arrival.\n";
  ev << "Queue length, # arrivals that saw n messages in \
  queue, estimated probability density function.\n";
  for(int i=0; i<jobDist->cells(); ++i)
  {   if(jobDist->cell(i) > 0)
      {   ev << i << ":\t" << jobDist->cell(i);
          ev << "\t" << jobDist->cellPDF(i) << endl;
      }
  }
  recordStats("Message Distribution Statistics", jobDist);
  ev << "Queue length statistics" << endl;
  num_samples = msgStat.samples();
  smallest = msgStat.min();
  largest = msgStat.max();
  mean = msgStat.mean();
  standard_deviation = msgStat.stddev(),
  variance = msgStat.variance();
  ev << "Number of samples: " << num_samples          << endl;
  ev << "Smallest queue: "    << smallest             << endl;
  ev << "Largest queue: "     << largest              << endl;
  ev << "Mean value: "        << mean                 << endl;
  ev << "Standar Dev: "       << standard_deviation   << endl;
  ev << "Variance: "          << variance             << endl;
  printf("Satellite: total frames received from GS=%d, \
  sent to SP=%d\n",
         framesReceivedFromGS, framesToPlanes);
  printf("Satellite: received from SP=%d, sent to GS=%d, \
  unrecognized=%d, in queue=%d\n",
  framesReceivedFromSP, framesToGS, unrecognized, queue.length());
}


//---------------------------------------------------------------
// File: simplebus.cc
// Based on an example by Andras Varga, author of OMNeT++.
//---------------------------------------------------------------

#include <assert.h>
#include <omnetpp.h>
```

```
#define MAX_NUM_TAPS  50

class SimpleBus : public cSimpleModule
{
    struct sTransmission
    {
        int tap, channel;
        bool upstream;
        bool isCollision;
        simtime_t busyStart, busyEnd;
        cMessage *endEvent;
        cMessage *frame;
    };

int prueba;
    Module_Class_Members(SimpleBus,cSimpleModule,0);
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);

    cMessage *createMessage();
    sTransmission *createTransmission();
    void recycleMessage(cMessage *msg);
    void recycleTransmission(sTransmission *tr);

  private:
    int numTaps;
    int numChannels;

    bool wantCollisionModeling;
    bool wantCollisionSignal;
    bool isFullDuplex;
    double delaySecPerMeter;
    double dataRateBps;

    char busTypePosition[20];
    double tapPositions[MAX_NUM_TAPS];
    cArray tapStates;

    cHead recycledMessages;
    cLinkedList recycledTransmissions;
};
```

```
Define_Module(SimpleBus);


void SimpleBus::initialize()
{
    // get parameters
// collision modeling flag
    wantCollisionModeling = par("wantCollisionModeling");
// "send collision signals" flag
    wantCollisionSignal = par("wantCollisionSignal");
// number of independent channels
    numChannels = par("numChannels");
// channel mode
    isFullDuplex = par("isFullDuplex");
// delay of the bus
    delaySecPerMeter = par("delaySecPerMeter");
// data rate of the bus
    dataRateBps = par("dataRateBps");
    strcpy(busTypePosition, par("busType").stringValue());
    strcat(busTypePosition, "position");
//  busTypePosition = LANposition, WPPposition, WSPposition,
//  or WGSposition

    // query the number of taps and the their positions (in meters)
    numTaps = gate("out")->size();
    assert(numTaps < MAX_NUM_TAPS);
    for (int k=0; k<numTaps; k++)
    {
        tapPositions[k] = gate("out",k)->toGate()->ownerModule()
        ->par(busTypePosition);
    }

    // create linked lists that will hold channel states at taps
    // (sTransmission structs)
    tapStates.setName("tapStates");
    for (int i=0; i<numTaps; i++)
    {
        for (int j=0; j<numChannels; j++)
        {
           char listname[64];
           sprintf(listname,"tap%dchannel%d",i,j);
           cLinkedList *list = new cLinkedList(listname);
```

```
            tapStates.addAt(i*numChannels+j, list);
        }
    }

    recycledMessages.setName("recycledMessages");
    recycledTransmissions.setName("recycledTransmissions");
}

cMessage *SimpleBus::createMessage()
{
    return new cMessage;
}

SimpleBus::sTransmission *SimpleBus::createTransmission()
{
    return new sTransmission;
}

void SimpleBus::recycleMessage(cMessage *msg)
{
    delete msg;
}

void SimpleBus::recycleTransmission(sTransmission *tr)
{
    delete tr;
}

void SimpleBus::handleMessage(cMessage *msg)
{
    cMessage *msg_new;

    // is msg a frame to be transmitted on the bus?
    if (!msg->isSelfMessage())
    {
        // get position where packet dropped in
        double packetPos = tapPositions[msg->arrivalGate()->index()];

        // get channel and direction of packet
        int channel = 0;
        if (msg->findPar("channel")>=0)
            channel = msg->par("channel");
```

```
        bool upstream = true;
        if (msg->findPar("upstream")>=0)
            upstream = msg->par("upstream");

        // duration of packet transmission
        double duration = msg->length() / dataRateBps;

        // check for collisions and schedule events at different taps
        for (int tap=0; tap<numTaps; tap++)
        {
// frame doesn't reach originating tap (J.V.)
// if channel is full duplex, frames propagate in only one
// direction, so maybe this frame won't reach this tap at all
            if ((packetPos == tapPositions[tap]) || isFullDuplex &&
               ((upstream && packetPos>tapPositions[tap]) ||
               (!upstream && packetPos<tapPositions[tap])))
                continue;

            // determine when frame head and tail will reach this tap
            double distance = fabs(packetPos-tapPositions[tap]);
            double delay = distance * delaySecPerMeter;

            simtime_t start = simTime() + delay;
            simtime_t end = start + duration;

            #ifdef WANT_DEBUG
              ev << "Start receive " << msg->name() << " at tap "
              << tap << " at T = " << start << endl;
              ev << "Complete receive " << msg->name() << "at tap "
              << tap << " at T = " << end << endl;
            #endif

            bool hasCollision = false;
            sTransmission *collisionTr = NULL;
            cLinkedList *list =
                (cLinkedList *)tapStates[tap*numChannels+channel];

            // if needed, do collision resolution at tap[tap]
            if (wantCollisionModeling)
            {
                for (cLinkedListIterator i(*list); !i.end(); i++)
                {
```

```
                    sTransmission *tr = (sTransmission *) i();

                    // does frame overlap with this transmission?
                    if (channel==tr->channel && (!isFullDuplex ||
                        upstream==tr->upstream) &&
                        end>tr->busyStart && start<tr->busyEnd)
                    {
//this is a collision; if we already had one, merge this transmission
// structure into the one already holding the collision, and discard
// this transmission struct.
                        if (hasCollision && tr!=collisionTr)
                        {
                            // extend (start,end) interval
                            if (start>tr->busyStart)
                                start = tr->busyStart;
                            if (end<tr->busyEnd)
                                end = tr->busyEnd;

                            // recycle this transmission
                           recycleMessage(cancelEvent(tr->endEvent));
                            if (tr->frame)
                                delete tr->frame;
                            list->remove(tr);
                            recycleTransmission(tr);

                            // adjust collisionTr afterwards...
                            tr = collisionTr;
                        }
                        else
                        {
                            // set collision flags
                            hasCollision = true;
                            collisionTr = tr;
                            tr->isCollision = true;

// if this transmission collided, don't need frame any more
                            if (tr->frame)
                            {
                                delete tr->frame;
                                tr->frame = NULL;
                            }
                        }
```

```
// adjust start and end times and reschedule events
                        if (tr->busyStart > start)
                            tr->busyStart = start;
                        else
                            start = tr->busyStart;

                        if (tr->busyEnd < end)
                        {
                            tr->busyEnd = end;
                          scheduleAt(end, cancelEvent(tr->endEvent));
                        }
                        else
                            end = tr->busyEnd;

                        #ifdef WANT_DEBUG
  ev << "*****CONTENT OF STRANSMISSION STRUCT AT TAP " << tap
     << " ******" << endl;
  ev << "channel = " << tr->channel << endl;
  ev << "tap = " << tr->tap << endl;
  ev << "busyStart = " << tr->busyStart << endl;
  ev << "busyEnd = " << tr->busyEnd << endl;
  ev << "**************************************************" <<endl;
                        #endif
                    }
                }
            }

// if no collision, add transmission structure and schedule
// associated events
            if (!hasCollision)
            {
                // create and fill in transmission structure
                sTransmission *tr = createTransmission();
                tr->tap = tap;
                tr->channel = channel;
                tr->upstream = upstream;
                tr->isCollision = false;
                tr->busyStart = start;
                tr->busyEnd = end;
                tr->frame = (cMessage *) msg->dup();
```

```
                // schedule event at end of transmission
                tr->endEvent = createMessage();
                char msgName[64];
                sprintf(msgName,"tap%dchannel%d-e",tap,channel);
                tr->endEvent->setName(msgName);
                tr->endEvent->setContextPointer(tr);
                scheduleAt(end, tr->endEvent);

                // add to list
                list->insertHead(tr);
            }
        }
        // don't need original frame any more
        delete msg;
    }

    else // msg->isSelfMessage() is true
    {
// this is a scheduled message, obtain associated
// transmission structure
        sTransmission *tr = (sTransmission *)msg->contextPointer();
        assert(msg==tr->endEvent);

// remove transmission structure from list
        cLinkedList *list = (cLinkedList *) tapStates[tr->
        tap*numChannels+tr->channel];
        list->remove(tr);

// send frame or collision signal on the corresponding tap
//this section changed so that collisions can be monitored
        if (tr->isCollision)
        {
             ev << "a collision signal output" << endl;
             if (wantCollisionSignal)
             {
                 msg_new= new cMessage("collision");
                 msg_new->setKind(1);
                 send(msg_new,"out",tr->tap);
                 ev<<busTypePosition[0]<<busTypePosition[1]<<
                 busTypePosition[2]<<
                 "bus:"<<simTime()<<" THE MESSAGE "<<
                 msg->name()<<" caused a collision"<<endl;
```

```
            }
        }
        else
        {
//             ev << "a signal output" << endl;
            msg_new=tr->frame;
            msg_new->setKind(2);
            send(msg_new, "out", tr->tap);
        }
        recycleTransmission(tr);
        recycleMessage(msg);
    }

}



#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define Dim 100
#define sigmoid
#define true  1
#define false 0

//                          GRADIENT DESCENT
// ---------------------global data structures ---------------------
// I is number of output nodes
// J is number of hidden nodes
// K is number of input nodes
// L is number of patterns processed for a weight update
// (periodic updates)
// L = 1 is continuous update, L = -1 is batch update.

 int I, J, K, L;
 double Alpha;  //Alpha is not used in this program
 double X[Dim], origX[Dim];
 double D[Dim];

 double W1[Dim][Dim];
```

```
 double Net1[Dim];
 double Y1[Dim];
 double Delta1[Dim];
 double DeltaW1[Dim][Dim];

 double W2[Dim][Dim];
 double Net2[Dim];
 double Y2[Dim];
 double Delta2[Dim];
 double DeltaW2[Dim][Dim];

 double W3[Dim][Dim];          //weights for Adaline network (J=0)
 double DeltaW3[Dim][Dim];

// ----------------------function prototypes-----------------------
double g (double x);
double gp(double x);
void   clearmat     (double W[Dim][Dim], int N, int M);
void   iniweights   (double W[Dim][Dim], int N, int M);
void   readweights  (FILE *fdw, double W[Dim][Dim], int N, int M);
void   writeweights (FILE *fdw, double W[Dim][Dim], int N, int M);
void   printmat     (double W[Dim][Dim], int N, int M);
void   printvec   (double V[Dim], int N);
void   printdata  (void);
int    readpat    (FILE *fdtrpr, double X[Dim], int K, double D[Dim],
                   int I, double origX[Dim]);
void   multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
               int N, int M);
double sqerror(FILE *fdtr, int *nok, int *nbad);
void   forward(void);
void   updateweights(void);
void   normalize(double V[Dim], int N);


// ======================MAIN PROGRAM============================
int main(void) {
 int i, j, k, p, ok, nok1, nok2, nbad1, nbad2;
 int epochs, predictedType, nextType, currentType;
 double sum, eta, epsilon, sq, sqOld;
 int stop2;
 double epsilon2;        // A second criterion to stop iterations.
 char justPrediction;    // T=training and prediction,
```

```
                        // P=skip training, only prediction
 FILE  *fdtr, *fdpr, *fdpa, *fdw;

// ------------------ execution starts here ---------------
// srand(time(NULL));
 srand(1);             // we want a fixed sequence of random weights
 sqOld = 0;
 stop2 = false;
 fdpa = fopen("params.pdu", "r");
 fdtr = fopen("training.pdu", "r");
 fdpr = fopen("predict.pdu", "r");
 fdw  = fopen("weights.pdu", "r");
 fscanf(fdpa, "%*[^:]: %d %d %d %lf %lf %lf %d %lf %c",
        &K, &J, &I, &eta, &epsilon, &Alpha, &L, &epsilon2,
        &justPrediction);
 assert(K>=1 && K<Dim);
 assert(J>=0 && J<Dim);
 assert(I>=1 && I<Dim);

 printf("Network parameters:\n");
 printf("  K=%d input  nodes (including bias)\n", K+1);
 if (J==0) printf("No hidden nodes. Adaline network assumed.\n");
    else   printf("  J=%d hidden nodes (including bias)\n", J+1);
 printf("  I=%d output nodes\n", I);
 printf(
 "Learning rate eta=%lf, stop criterion epsilon=%lf\n", eta,epsilon);
 printf("Weight update every L=%d patterns\n", L);
 printf("Second stop criterion epsilon2=%lf\n", epsilon2);
 printf("justPrediction=%c\n",justPrediction);
 if (fdw == NULL) printf("No weights.pdw file exists\n");
 else             printf ("Weights.pdw file opened\n");
 getchar();
 epochs = 0;

// Step 1: initialize the weights.
 printf("Initializing weights ...\n");
 if (J==0) {
   if (fdw == NULL) iniweights(W3, I, K);
   else             readweights(fdw, W3, I, K);
   clearmat(DeltaW3, I, K);
 }
 else {
```

```
   if (fdw == NULL) { iniweights(W1, J, K);
                      iniweights(W2, I, J);
   }
   else { printf("About to read weights\n");
          readweights(fdw, W1, J, K);
          readweights(fdw, W2, I, J);
          printf("Weights read\n");
   }
   clearmat(DeltaW1, J, K);
   clearmat(DeltaW2, I, J);
 }
 if (fdw != NULL) { fclose(fdw); printf("Weights file closed\n");}
 printf("Weights are ready!\n");
 if (justPrediction == 'P') goto predict;

// Step 2: present an input pattern from the training collection
printf("Starting Training Phase\n");
step2:
 rewind(fdtr);
 p = 0;
 while (readpat(fdtr, X, K, D, I, origX) != EOF) {
   p++;

// Step 3: calculate outputs of nodes (hidden and output layer)
   forward();

// Step 4: check to see whether Y2[i] = D[i]
// This step is not implemented because it is unnecessary.

// Step 5:  calculate the error terms in output and hidden layers
   if (J==0) {
     for (i=1; i<=I; i++)
       Delta2[i] = D[i]-Y2[i];  //Delta for Adaline network
   }
   else {
     for (i=1; i<=I; i++)
       Delta2[i] = gp(Net2[i]) * (D[i]-Y2[i]);
     for (j=1; j<=J; j++) {
       sum = 0.0;
       for (i=1; i<=I; i++)
         sum += W2[i][j] * Delta2[i];
       Delta1[j] = gp(Net1[j]) * sum;
```

```
     }
   }

// Step 5bis: accumulate errors for periodic update
   if (J==0) {
     for (i=1; i<=I; i++)
     for (k=0; k<=K; k++)
       DeltaW3[i][k] += eta * Delta2[i] * X[k];    //Adaline weights
   }
   else {
     for (i=1; i<=I; i++)
     for (j=0; j<=J; j++)
       DeltaW2[i][j] += eta * Delta2[i] * Y1[j];

     for (j=1; j<=J; j++)
     for (k=0; k<=K; k++)
       DeltaW1[j][k] += eta * Delta1[j] * X[k];
   }

//if (epochs==0) printdata();

// Step 6: change the weights if periodic update applies
   if ((L>0) && (p%L == 0))             // Periodic updates apply?
      updateweights();

// Step 7: Are there more patterns,
// is the convergence criterion satisfied?
step7:
 } /*end of step2: while there are more patterns*/

// Last weight update for this epoch if applies
 if ((L<=0) || (p%L != 0))             // Last update apply?
    updateweights();
 epochs++;
 sq = sqerror(fdtr, &nok1, &nbad1);
 if (epochs%10 == 0) {
    printf("Epochs = %d, sqerror = %lf, nok1=%d, nbad1=%d\n",
            epochs, sq, nok1, nbad1);
    if (epochs%1000 == 0)
      if (fabs(sqOld-sq) < epsilon2 ) stop2 = true;
      else sqOld = sq;
 }
```

```
 if (sq > epsilon && !stop2) goto step2;

 printf("Training complete!\n");
// printdata();
 fdw  = fopen("weights.pdu", "w");
 if (J==0) writeweights(fdw, W3, I, K);
 else { writeweights(fdw, W1, J, K);
        writeweights(fdw, W2, I, J);
 }
 fclose(fdw);

predict:
 printf("Starting Prediction Phase\n");
// Now predictions will be tested for data in the predict.dat file
 p = nok1 = nok2 = nbad1 = nbad2 = 0;
 int nok3=0, nbad3=0, decisionWait2 = 0, decisionSend2 = 0,
     decisionWait3 = 0, decisionSend3 = 0;
//reading predictions "predict.pdu". origX = non-normalized copy of X
 while (readpat(fdpr, X, K, D, I, origX) != EOF)
 {
   p++;
//printf("origX[%d]=%lf\n", K, origX[K]);
   currentType = origX[K];
   nextType = 0;
   for (i=I; i>=1; i--) nextType = nextType*2 + D[i];
//printf("Line #%d, currentType %d, nextType %d\n",
//        p, currentType, nextType);

//   printf("New prediction p=%d being read... X= ", p);
//   printvec(X, K);
   forward();
/*   printf("Y2 = ");
   printvec(Y2, I);
   printf("D = ");
   printvec(D, I);
*/
#ifdef sigmoid
   predictedType = 0;
   for (i=I; i>=1; i--) predictedType = predictedType*2 +
       (int)trunc(Y2[i]+0.5);
   ok = 1;
   for (i=1; i<=I && ok == 1; i++)
```

```
        if ((D[i]-0.5)*(Y2[i]-0.5)<=0.) ok = 0;
    if (ok==1) nok1++;
    else nbad1++;
    if ( ((nextType!=currentType) && (predictedType!=currentType)) ||
           (nextType==currentType) && (predictedType == currentType) )
            nok2++;
    else nbad2++;
    if (nextType == predictedType) nok3++;
    else nbad3++;
// nok1 = number of correcttly predicted PDUs
// nok2 = number of correct decisions based on predictions
// nok3 = number of correcttly predicted PDUs = nok1

    if (predictedType == currentType) decisionWait2++;
    else decisionSend2++;
    if (nextType == currentType) decisionWait3++;
    else decisionSend3++;
//printf("current=%d, next=%d, predict=%d, nok1=%d, nbad1=%d,nok2=%d,
//          nbad2=%d, nok3=%d, nbad3=%d, W2=%d, S2=%d, W3=%d, S3=%d\n",
//          currentType, nextType, predictedType, nok1, nbad1, nok2,
//          nbad2, nok3, nbad3, decisionWait2, decisionSend2,
//          decisionWait3, decisionSend3);
#endif

#ifdef tanh
    ok = 1;
    for (i=1; i<=I && ok == 1; i++)
        if (D[i]*Y2[i]<=0.) ok = 0;
    if (ok==1) nok1++;
    else nbad1++;
#endif
 } //EOF on predictions file "predict.pdu"

 printf("Predictions correct=%d, bad=%d total=%d\n", nok1, nbad1, p);
 printf("Predictions2 correct=%d, bad=%d\n", nok2, nbad2);
 printf("Predictions3 correct=%d, bad=%d\n", nok3, nbad3);
 printf("Predictions2 Wait=%d, Send=%d\n",
        decisionWait2, decisionSend2);
 printf("Predictions3 Wait=%d, Send=%d\n",
        decisionWait3, decisionSend3);

 fclose(fdpa);
```

```
 fclose(fdtr);
 fclose(fdpr);
 return 0;
}

// ------------------------------------------------------------------
void clearmat (double W[Dim][Dim], int N, int M){
 int i, j;
 for (i=0; i<=N; i++)
 for (j=0; j<=M; j++)
   W[i][j] = 0.0;
}

// ------------------------------------------------------------------
void iniweights (double W[Dim][Dim], int N, int M) {
 int i, j;
 for (j=0; j<=M; j++) {
   W[0][j] = 0.0;             // These entries are not really used
   for (i=1; i<=N; i++) {
     W[i][j] = (double)rand()/(double)RAND_MAX - 0.5;
   }
 }
}

// ------------------------------------------------------------------
void readweights (FILE *fdw, double W[Dim][Dim], int N, int M) {
 int i, j, Nw, Mw;
 fscanf(fdw, ": %d %d\n", &Nw, &Mw);
 assert(Nw==N && Mw==M);
 for (i=0; i<=N; i++) {
   for (j=0; j<=M; j++) fscanf(fdw, "%lE ", &W[i][j]);
   fscanf(fdw, "\n");
 }
}

// ------------------------------------------------------------------
void writeweights (FILE *fdw, double W[Dim][Dim], int N, int M) {
 int i, j;
 fprintf(fdw, ": %d %d\n", N, M);
 for (i=0; i<=N; i++) {
   for (j=0; j<=M; j++) fprintf(fdw, "%25.15lE ", W[i][j]);
   fprintf(fdw, "\n");
```

```
 }
}

// ----------------------------------------------------------------
void updateweights(void) {
 int i, j, k;
 if (J==0) {
    for (i=1; i<=I; i++)
     for (k=0; k<=K; k++)
      W3[i][k] += DeltaW3[i][k];    //Adaline weights
    clearmat(DeltaW3, I, K);
 }
 else {
    for (i=1; i<=I; i++)
     for (j=0; j<=J; j++)
      W2[i][j] += DeltaW2[i][j];

    for (j=1; j<=J; j++)
     for (k=0; k<=K; k++)
      W1[j][k] += DeltaW1[j][k];
    clearmat(DeltaW1, J, K);
    clearmat(DeltaW2, I, J);
 }
}
// ----------------------------------------------------------------
#ifdef sigmoid
double g(double x) {    //sigmoid activation function
  double r;
  if (x > 50.0) r = 1.0;
  else if (x < -50.0) r = 0.0;
       else {
         r = 1.0 /(1.0 + exp(-x));
       }
//  if(r==1.0 || r==0.0) printf("Warning in sigmoid activation
//  function: x=%lf\n", x);
  return r;
}

// ----------------------------------------------------------------
double gp(double x) {
  double r;
  r = g(x);
```

```
  return r*(1.0 - r);
}
#endif

// ------------------------------------------------------------------
#ifdef tanh
double g(double x) {    //hyperbolic tangent activation function
  double r;
  if (x > 50.0) r = 1.0;
  else if (x < -50.0) r = -1.0;
       else {
         r = exp(-2.0*x);
         r = (1.0 - r)/(1.0 + r);
       }
  if(r==1.0 || r==0.0)
    printf("Warning in tanh activation function: x=%lf\n", x);
  return r;
}

// ------------------------------------------------------------------
double gp(double x) {
  double tmp;
  tmp = g(x);
  return 1.0 - tmp*tmp;
}
#endif

// ------------------------------------------------------------------
void printmat (double W[Dim][Dim], int N, int M) {
 int i, j;

 for (i=1; i<=N; i++) {
   printf("[%d,*]: ", i);
   for (j=0; j<=M; j++)
     printf ("%7.3lf ", W[i][j]);
   printf("\n");
 }
}

// ------------------------------------------------------------------
void printvec (double V[Dim], int N) {
 int i;
```

```
 for (i=1; i<=N; i++) printf("%7.3lf ", V[i]);
 printf("\n");
}

// -----------------------------------------------------------------
void printdata (void) {
   printf("Printdata: X     : ");     printvec(X, K);
   if (J==0) {
     printf("Printdata: W3:\n");      printmat(W3, I, K);
     printf("Printdata: DeltaW3:\n"); printmat(DeltaW3, I, K);
   }
   else {
     printf("Printdata: W1:\n");      printmat(W1, J, K);
     printf("Printdata: DeltaW1:\n"); printmat(DeltaW1, J, K);
     printf("Printdata: Net1  : ");   printvec(Net1, J);
     printf("Printdata: Y1    : ");   printvec(Y1, J);
     printf("Printdata: Delta1: ");   printvec(Delta1, J);
     printf("Printdata: W2\n");       printmat(W2, I, J);
     printf("Printdata: DeltaW2:\n"); printmat(DeltaW2, I, J);
   }
   printf("Printdata: Net2  : ");     printvec(Net2, I);
   printf("Printdata: Y2    : ");     printvec(Y2, I);
   printf("Printdata: Delta2: ");     printvec(Delta2, I);
   printf("Printdata: D     : ");     printvec(D, I);
   printf("\n");
}

// -----------------------------------------------------------------
int readpat (FILE *fdtrpr, double X[Dim], int K, double D[Dim],
    int I, double origX[Dim]) {
 int i, k, eof;
 X[0] = 1.0; origX[0] = 1.0;
 for (k=1; k<=K; k++)  {
 // reading from "training.pdu" or "predict.pdu" file
   fscanf(fdtrpr, "%lf", &X[k]);
   origX[k] = X[k];             // saves a non-normalized copy of X.
 }
 normalize(X, K);
 D[0] = 0.0;
 for (i=1; i<=I; i++)  eof = fscanf(fdtr, "%lf", &D[i]);
 return eof;
}
```

```
// ----------------------------------------------------------------
void normalize(double V[Dim], int N) {
 int i;
 double norm = 0.0;
 for (i=0; i<=N; i++)
   norm += fabs(V[i]);
 if (norm>0)
   for (i=0; i<=N; i++)
     V[i] /= norm;
 return;
}

// ----------------------------------------------------------------
void multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
             int N, int M)
{
 int i, j;
 B[0] = 0.0;
 for (i=1; i<=N; i++) {
   B[i] = 0.0;
   for (j=0; j<=M; j++)
     B[i] += A[i][j] * X[j];
 }
}

// ----------------------------------------------------------------
void forward(){
 int i,j;

 if (J==0) {
   multAX (W3, X, Net2, I, K);
   Y2[0] = 0.0;                  //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = Net2[i];            //g(Net2[i]) = Net2[i]
 }
 else {
   multAX (W1, X, Net1, J, K);
   Y1[0] = 1.0;                  //bias node
   for (j=1; j<=J; j++)
     Y1[j] = g(Net1[j]);
   multAX (W2, Y1, Net2, I, J);
```

```
   Y2[0] = 0.0;                    //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = g(Net2[i]);
 }
}

// -----------------------------------------------------------------
double sqerror (FILE *fdtr, int *nok, int *nbad) {
 int i, pt, ok;
 double err, tmp;

 err = 0.0;
 pt = 0;
 rewind (fdtr);
 (*nok) = (*nbad) = 0;
 while ( readpat(fdtr, X, K, D, I, origX) != EOF) {
   pt++;
   forward();
   ok = 1;
   for (i=1; i<=I; i++) {
     tmp = fabs(D[i] - Y2[i]);
     err += tmp*tmp;
     #ifdef sigmoid
     if ((D[i]-0.5)*(Y2[i]-0.5)<=0.) ok = 0;
     #endif
     #ifdef tanh
     if (D[i]*Y2[i]<=0.) ok = 0;
     #endif
   }
   if (ok==1) (*nok)++;
   else (*nbad)++;
 }
 err /= (2.0 * pt);
 return err;
}

// -----------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
```

```
#define Dim 100
#define sigmoid
#define true  1
#define false 0

//                     BACK-PROPAGATION WITH MOMENTUM
// --------------------global data structures --------------------
// I is number of output nodes
// J is number of hidden nodes
// K is number of input nodes
// L is number of patterns processed for a weight update
// (periodic updates)
// L = 1 is continuous update, L = -1 is batch update.

 int I, J, K, L;
 double Alpha;
 double X[Dim];
 double D[Dim];

 double W1[Dim][Dim];
 double Net1[Dim];
 double Y1[Dim];
 double Delta1[Dim];
 double DeltaW1[Dim][Dim];

 double W2[Dim][Dim];
 double Net2[Dim];
 double Y2[Dim];
 double Delta2[Dim];
 double DeltaW2[Dim][Dim];

 double W3[Dim][Dim];          //weights for Adaline network (J=0)
 double DeltaW3[Dim][Dim];

// ----------------------function prototypes----------------------
double g (double x);
double gp(double x);
void   clearmat   (double W[Dim][Dim], int N, int M);
void   iniweights (double W[Dim][Dim], int N, int M);
void   readweights  (FILE *fdw, double W[Dim][Dim], int N, int M);
void   writeweights (FILE *fdw, double W[Dim][Dim], int N, int M);
void   printmat   (double W[Dim][Dim], int N, int M);
```

```
void   printvec   (double V[Dim], int N);
void   printdata  (void);
int    readpat    (FILE *fdtr, double X[Dim], int K, double D[Dim],
                   int I);
void   multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
               int N, int M);
double sqerror(FILE *fdtr, int *nok, int *nbad);
void   forward(void);
void   updateweights(void);
void   normalize(double V[Dim], int N);

// =====================MAIN PROGRAM============================
int main(void) {
 int i, j, k, p, ok, nok, nbad;
 int epochs;
 double sum, eta, epsilon, sq, sqOld;
 int stop2;
 double epsilon2;        // A second criterion to stop iterations.
// T=training and prediction, P=skip training, only prediction
 char justPrediction;
 FILE  *fdtr, *fdpr, *fdpa, *fdw;

// ------------------ execution starts here ---------------
// srand(time(NULL));
 srand(1);              // we want a fixed sequence of random weights
 sqOld = 0;
 stop2 = false;
 fdpa = fopen("params.pdu", "r");
 fdtr = fopen("training.pdu", "r");
 fdpr = fopen("predict.pdu", "r");
 fdw  = fopen("weights.pdu", "r");
 fscanf(fdpa, "%*[^:]: %d %d %d %lf %lf %lf %d %lf %c",
              &K, &J, &I, &eta, &epsilon, &Alpha, &L,
              &epsilon2, &justPrediction);
 assert(K>=1 && K<Dim);
 assert(J>=0 && J<Dim);
 assert(I>=1 && I<Dim);

 printf("Network parameters:\n");
 printf("  K=%d input  nodes (including bias)\n", K+1);
 if (J==0) printf("No hidden nodes. Adaline network assumed.\n");
    else   printf("  J=%d hidden nodes (including bias)\n", J+1);
```

```
 printf("  I=%d output nodes\n", I);
 printf("Learning rate eta=%lf, stop criterion epsilon=%lf,"
        "alpha=%lf\n", eta, epsilon, Alpha);
 printf("Weight update every L=%d patterns\n", L);
 printf("Second stop criterion epsilon2=%lf\n", epsilon2);
 if (fdw == NULL) printf("No weights.pdw file exists\n");
 else             printf ("Weights.pdw file opened\n");
 getchar();
 epochs = 0;

// Step 1: initialize the weights.
 printf("Initializing weights ...\n");
 if (J==0) {
   if (fdw == NULL) iniweights(W3, I, K);
   else             readweights(fdw, W3, I, K);
   clearmat(DeltaW3, I, K);
 }
 else {
   if (fdw == NULL) { iniweights(W1, J, K);
                      iniweights(W2, I, J);
   }
   else { printf("About to read weights\n");
          readweights(fdw, W1, J, K);
          readweights(fdw, W2, I, J);
          printf("Weights read\n");
   }
   clearmat(DeltaW1, J, K);
   clearmat(DeltaW2, I, J);
 }
 if (fdw != NULL) { fclose(fdw); printf("Weights file closed\n");}
 printf("Weights are ready!\n");

// Step 2: present an input pattern from the training collection
step2:
 rewind(fdtr);
 p = 0;

 while (readpat(fdtr, X, K, D, I) != EOF) {
   p++;

// Step 3: calculate outputs of nodes (hidden and output layer)
   forward();
```

```
// Step 4: check to see whether Y2[i] = D[i]
// This step is not implemented because it is unnecessary.

// Step 5:  calculate the error terms in output and hidden layers
   if (J==0) {
     for (i=1; i<=I; i++)
       Delta2[i] = D[i]-Y2[i];  //Delta for Adaline network
   }
   else {
     for (i=1; i<=I; i++)
       Delta2[i] = gp(Net2[i]) * (D[i]-Y2[i]);
     for (j=1; j<=J; j++) {
       sum = 0.0;
       for (i=1; i<=I; i++)
         sum += W2[i][j] * Delta2[i];
       Delta1[j] = gp(Net1[j]) * sum;
     }
   }

// Step 5bis: accumulate errors for periodic update
   if (J==0) {
     for (i=1; i<=I; i++)
     for (k=0; k<=K; k++)
       DeltaW3[i][k] += eta * Delta2[i] * X[k];    //Adaline weights
   }
   else {
     for (i=1; i<=I; i++)
     for (j=0; j<=J; j++)
       DeltaW2[i][j] += eta * Delta2[i] * Y1[j];

     for (j=1; j<=J; j++)
     for (k=0; k<=K; k++)
       DeltaW1[j][k] += eta * Delta1[j] * X[k];
   }

//if (epochs==0) printdata();

// Step 6: change the weights if periodic update applies
   if ((L>0) && (p%L == 0))             // Periodic updates apply?
      updateweights();
```

```
// Step 7: Are there more patterns,
// is the convergence criterion satisfied?
step7:
 } /*end of step2: while there are more patterns*/

// Last weight update for this epoch if applies
 if ((L<=0) || (p%L != 0))             // Last update apply?
    updateweights();
 epochs++;
 sq = sqerror(fdtr, &nok, &nbad);
if (epochs%10 == 0) {
    printf("Epochs = %d, sqerror = %lf, nok=%d, nbad=%d\n",
            epochs, sq, nok, nbad);
    if (epochs%1000 == 0)
      if (fabs(sqOld-sq) < epsilon2 ) stop2 = true;
      else sqOld = sq;
 }
 if (sq > epsilon && !stop2) goto step2;

 printf("Training complete!\n");
// printdata();
 fdw  = fopen("weights.pdu", "w");
 if (J==0) writeweights(fdw, W3, I, K);
 else { writeweights(fdw, W1, J, K);
        writeweights(fdw, W2, I, J);
 }
 fclose(fdw);

// Now predictions will be tested for data in the predict.dat file
 p = nok = nbad = 0;
 while (readpat(fdpr, X, K, D, I) != EOF) {
   p++;
//   printf("New prediction p=%d being read... X= ", p);
//   printvec(X, K);
   forward();
/*   printf("Y2 = ");
   printvec(Y2, I);
   printf("D = ");
   printvec(D, I);
*/
#ifdef sigmoid
     ok = 1;
```

```
     for (i=1; i<=I && ok == 1; i++)
       if ((D[i]-0.5)*(Y2[i]-0.5)<=0.) ok = 0;
     if (ok==1) nok++;
     else nbad++;
#endif

#ifdef tanh
     ok = 1;
     for (i=1; i<=I && ok == 1; i++)
       if (D[i]*Y2[i]<=0.) ok = 0;
     if (ok==1) nok++;
     else nbad++;
#endif

 }

 printf("Predictions correct=%d, bad=%d total=%d\n", nok, nbad, p);
 fclose(fdpa);
 fclose(fdtr);
 fclose(fdpr);
 return 0;
}

// ----------------------------------------------------------------
void   clearmat   (double W[Dim][Dim], int N, int M){
 int i, j;
 for (i=0; i<=N; i++)
 for (j=0; j<=M; j++)
   W[i][j] = 0.0;
}

// ----------------------------------------------------------------
void iniweights (double W[Dim][Dim], int N, int M) {
 int i, j;
 double scale = 1.000;          // To control size of random values.
 for (j=0; j<=M; j++) {
   W[0][j] = 0.0;              // These entries are not really used
   for (i=1; i<=N; i++) {
     W[i][j] = ((double)rand()/(double)RAND_MAX - 0.5)*scale;
   }
 }
}
```

```
// ----------------------------------------------------------------
void readweights (FILE *fdw, double W[Dim][Dim], int N, int M) {
 int i, j, Nw, Mw;
 fscanf(fdw, ": %d %d\n", &Nw, &Mw);
 assert(Nw==N && Mw==M);
 for (i=0; i<=N; i++) {
   for (j=0; j<=M; j++) fscanf(fdw, "%lE ", &W[i][j]);
   fscanf(fdw, "\n");
 }
}

// ----------------------------------------------------------------
void writeweights (FILE *fdw, double W[Dim][Dim], int N, int M) {
 int i, j;
 fprintf(fdw, ": %d %d\n", N, M);
 for (i=0; i<=N; i++) {
   for (j=0; j<=M; j++) fprintf(fdw, "%25.15lE ", W[i][j]);
   fprintf(fdw, "\n");
 }
}

// ----------------------------------------------------------------
void   updateweights(void) {
 int i, j, k;
 if (J==0) {
    for (i=1; i<=I; i++)
     for (k=0; k<=K; k++) {
      W3[i][k] += DeltaW3[i][k];            //Adaline weights
      DeltaW3[i][k] *= Alpha;
    }
 }
 else {
    for (i=1; i<=I; i++)
     for (j=0; j<=J; j++) {
      W2[i][j] += DeltaW2[i][j];
      DeltaW2[i][j] *= Alpha;
    }
    for (j=1; j<=J; j++)
     for (k=0; k<=K; k++) {
      W1[j][k] += DeltaW1[j][k];
      DeltaW1[j][k] *= Alpha;
```

```
    }
  }
}
// ------------------------------------------------------------------
#ifdef sigmoid
double g(double x) {    //sigmoid activation function
  double r;
  if (x > 50.0) r = 1.0;
  else if (x < -50.0) r = 0.0;
       else {
         r = 1.0 /(1.0 + exp(-x));
       }
//  if(r==1.0 || r==0.0)
//  printf("Warning in sigmoid activation function: x=%lf\n", x);
  return r;
}

double gp(double x) {
  double r;
  r = g(x);
  return r*(1.0 - r);
}
#endif

// ------------------------------------------------------------------
#ifdef tanh
double g(double x) {    //hyperbolic tangent activation function
  double r;
  if (x > 50.0) r = 1.0;
  else if (x < -50.0) r = -1.0;
       else {
         r = exp(-2.0*x);
         r = (1.0 - r)/(1.0 + r);
       }
  if(r==1.0 || r==0.0)
    printf("Warning in tanh activation function: x=%lf\n", x);
  return r;
}

double gp(double x) {
  double tmp;
  tmp = g(x);
```

```
  return 1.0 - tmp*tmp;
}
#endif

// -----------------------------------------------------------------
void printmat (double W[Dim][Dim], int N, int M) {
 int i, j;

 for (i=1; i<=N; i++) {
   printf("[%d,*]: ", i);
   for (j=0; j<=M; j++)
     printf ("%7.3lf ", W[i][j]);
   printf("\n");
 }
}

// -----------------------------------------------------------------
void printvec (double V[Dim], int N) {
 int i;
 for (i=1; i<=N; i++) printf("%7.3lf ", V[i]);
 printf("\n");
}

// -----------------------------------------------------------------
void printdata (void) {
   printf("Printdata: X     : ");     printvec(X, K);
   if (J==0) {
     printf("Printdata: W3:\n");      printmat(W3, I, K);
     printf("Printdata: DeltaW3:\n"); printmat(DeltaW3, I, K);
   }
   else {
     printf("Printdata: W1:\n");      printmat(W1, J, K);
     printf("Printdata: DeltaW1:\n"); printmat(DeltaW1, J, K);
     printf("Printdata: Net1  : ");   printvec(Net1, J);
     printf("Printdata: Y1    : ");   printvec(Y1, J);
     printf("Printdata: Delta1: ");   printvec(Delta1, J);
     printf("Printdata: W2\n");       printmat(W2, I, J);
     printf("Printdata: DeltaW2:\n"); printmat(DeltaW2, I, J);
   }
   printf("Printdata: Net2  : ");     printvec(Net2, I);
   printf("Printdata: Y2    : ");     printvec(Y2, I);
   printf("Printdata: Delta2: ");     printvec(Delta2, I);
```

```
   printf("Printdata: D     : ");     printvec(D, I);
   printf("\n");
}

// -----------------------------------------------------------------
int readpat (FILE *fdtr, double X[Dim], int K, double D[Dim], int I)
{
 int i, k, eof;
 X[0] = 1.0;
 for (k=1; k<=K; k++)  fscanf(fdtr, "%lf", &X[k]);
 normalize(X, K);
 D[0] = 0.0;
 for (i=1; i<=I; i++)  eof = fscanf(fdtr, "%lf", &D[i]);
 return eof;
}

// -----------------------------------------------------------------
void normalize(double V[Dim], int N) {
 int i;
 double norm = 0.0;
 for (i=0; i<=N; i++)
   norm += fabs(V[i]);
 if (norm>0)
   for (i=0; i<=N; i++)
     V[i] /= norm;
 return;
}

// -----------------------------------------------------------------
void multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
             int N, int M)
{
 int i, j;
 B[0] = 0.0;
 for (i=1; i<=N; i++) {
   B[i] = 0.0;
   for (j=0; j<=M; j++)
     B[i] += A[i][j] * X[j];
 }
}

// -----------------------------------------------------------------
```

```
void forward(){
 int i,j;

 if (J==0) {                          //Adaline network
   multAX (W3, X, Net2, I, K);
   Y2[0] = 0.0;                       //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = Net2[i];                 //g(Net2[i]) = Net2[i]
 }
 else {
   multAX (W1, X, Net1, J, K);
   Y1[0] = 1.0;                       //bias node
   for (j=1; j<=J; j++)
     Y1[j] = g(Net1[j]);
   multAX (W2, Y1, Net2, I, J);
   Y2[0] = 0.0;                       //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = g(Net2[i]);
 }
}

// -----------------------------------------------------------------
double sqerror (FILE *fdtr, int *nok, int *nbad) {
 int i, pt, ok;
 double err, tmp;

 err = 0.0;
 pt = 0;
 rewind (fdtr);
 (*nok) = (*nbad) = 0;
 while ( readpat(fdtr, X, K, D, I) != EOF) {
   pt++;
   forward();
   ok = 1;
   for (i=1; i<=I; i++) {
     tmp = fabs(D[i] - Y2[i]);
     err += tmp*tmp;
     #ifdef sigmoid
     if ((D[i]-0.5)*(Y2[i]-0.5)<=0.) ok = 0;
     #endif
     #ifdef tanh
     if (D[i]*Y2[i]<=0.) ok = 0;
```

```
     #endif
   }
   if (ok==1) (*nok)++;
   else (*nbad)++;
 }
 err /= (2.0 * pt);
 return err;
}

// ---------------------------------------------------------------

// This program reads a data.pdu file containing sequences of
// consecutive numeric PDUs and the corresponding next binary
// PDU predicted.
// The program creates two random partitions out of the file,
// intended for training and testing a Neural Network.

#include <stdio.h>
#include <stdlib.h>
#define LN 1000
FILE * fdIn, * fdOut1, * fdOut2;
int i, j, numLines, lineSize, fileSize, halfLines;
int * record, recordLimit;
char line[LN];
int main() {
// srand (time (0)); rand();
 fdIn = fopen ("data0N45.pdu", "r");
 fdOut1 = fopen ("training.pdu", "w");
 fdOut2 = fopen ("predict.pdu", "w");
 fgets(line, LN, fdIn);
 lineSize = strlen(line)+1;

 fseek(fdIn, 0, SEEK_END);
 fileSize = ftell(fdIn);
 numLines = fileSize / lineSize;
 halfLines = numLines/2;
 record = (int *) malloc(sizeof(int) * numLines);
 for (i=0; i<numLines; i++) record[i] = i;

 recordLimit = numLines-1;
 for (i=0; i<halfLines; i++) {
   j = rand()*recordLimit / RAND_MAX;
```

```
   fseek(fdIn, record[j]*lineSize, SEEK_SET);
   fgets(line, LN, fdIn);
   fputs(line, fdOut1);
//  printf("fdOut1: iteration=%d, random j = %d, index=%d, line=%s",
//   i, j, record[j], line);
//   getchar();
   record[j] = record[recordLimit];
   recordLimit--;
 }
 while (recordLimit >= 0) {
   j = rand()*recordLimit / RAND_MAX;
   fseek(fdIn, record[j]*lineSize, SEEK_SET);
   fgets(line, LN, fdIn);
   fputs(line, fdOut2);
//   printf("fdOut2: iteration=%d, random j = %d, index=%d, line=%s",
//   i, j, record[j], line);
//   getchar();
   record[j] = record[recordLimit];
   recordLimit--;
 }
 fclose (fdIn);
 fclose (fdOut1);
 fclose (fdOut2);
}


// This program reads in dataNN.txt containing the summary PDUs and
// creates a new summary file called dataNewNN.txt containing the
// prediction 'W' (wait) or 'S' (send) + the old data
// The program reads in the weights calculated by gd.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define Dim 100
#define sigmoid
#define true  1
#define false 0

// ---------------------global data structures ---------------------
// I is number of output nodes
```

```
// J is number of hidden nodes
// K is number of input nodes

 int I, J, K, L, sizePDUtable;
 unsigned int prevTimePDU, prevLengthPDU, currTimePDU, currLengthPDU;
 double D[Dim], Delta1[Dim], Delta2[Dim], DeltaW1[Dim][Dim],
        DeltaW2[Dim][Dim], DeltaW3[Dim][Dim], Net1[Dim],  Net2[Dim],
        origX[Dim],        W1[Dim][Dim],      W2[Dim][Dim], X[Dim],
        Y1[Dim],           Y2[Dim];

 double W3[Dim][Dim];          //weights for Adaline network (J=0)
 char   buf[100], typePDU[100],
        *PDUtable[]= {               // table to store all PDU types
                "",                      // 0 not used
                "laser",                 // 1
                "start_resume",          // 2
                "stop_freeze",           // 3
                "po_task_authorization", // 4
                "po_minefield",          // 5
                "fire",                  // 6
                "detonation",            // 7
                "acknowledge",           // 8
                "po_delete_objects",     // 9
                "minefield",             // 10
                "po_message",            // 11
                "signal",                // 12
                "aggregate_state",       // 13
                "po_simulator_present",  // 14
                "po_task_frame",         // 15
                "mines",                 // 16
                "po_point",              // 17
                "po_objects_present",    // 18
                "po_fire_parameters",    // 19
                "iff",                   // 20
                "po_line",               // 21
                "po_parametric_input",   // 22
                "po_unit",               // 23
                "po_task",               // 24
                "transmitter",           // 25
                "po_task_state",         // 26
                "entity_state",          // 27
                "" }                     // 28 not used
```

```
;
 FILE  *fdpa, *fdw, *fdpr, *fdnew;

// ----------------------function prototypes-----------------------
double g (double x);
void clearmat     (double W[Dim][Dim], int N, int M);
void readweights  (FILE *fdw, double W[Dim][Dim], int N, int M);
void printdata  (void);
int  readpat (double X[Dim], int K, double D[Dim], int I,
              double origX[Dim]);
void multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
             int N, int M);
void forward(void);
void normalize(double V[Dim], int N);

// ======================MAIN PROGRAM============================
int main(int argc, char *argv[]) {
 int i, p;
 int decisionWait, decisionSend;
 int predictedType, currentType;
 char predict[20], dataNew[20];

// ------------------ execution starts here ---------------
 if (argc < 2)
  {
   printf("Usage: %s <dataNN.txt>\n", argv[0]);
   return 1;
  }
  sizePDUtable = sizeof(PDUtable)/sizeof(char *);
  for (I=0; I<Dim; I++) {
   D[I]=Delta1[I]=Delta2[I]=Net1[I]=Net2[I]=origX[I]=0.;
   X[I]=Y1[I]=Y2[I]=0.;
  }

 strcpy(predict, argv[1]);
 //e.g. predict = "data3.pdu"
 strncpy(dataNew, predict, 4); dataNew[4]='\0';
 //e.g. dataNew = "data"
 strcat(dataNew, "New"); strcat(dataNew, &predict[4]);
 //dataNew = "dataNew3.pdu"

 fdpa = fopen("C:\\PhD\\NNPDUPred\\params.pdu", "r");
```

```
 // params are K,J,I. The rest (eta,epsilon,
 //Alpha,L,epsilon2,justPrediction) is ignored
 fdw  = fopen("C:\\PhD\\NNPDUPred\\weights.pdu", "r");
 // weights calculated by gd.c
 fdpr = fopen(predict, "r");
 // opens "data3.pdu" for reading. Sample data is:
 //0x55f17462    32 | :20:08.575   1 <dis204 acknowledge PDU>: 18
 //0x5602dbaa    32 | :20:09.531   2 <dis204 acknowledge PDU>: 72
 //0x5614fd5c    32 | :20:10.527   3 <dis204 acknowledge PDU>: 106

 fdnew = fopen(dataNew, "w");   // opens "dataNew3.pdu" for writing
 fscanf(fdpa, "%*[^:]: %d %d %d", &K, &J, &I); // fdpa = "params.pdu"

 assert(K>=1 && K<Dim);
 assert(J>=0 && J<Dim);
 assert(I>=1 && I<Dim);
 assert (fdw != NULL);

 printf("Network parameters:\n");
 printf("  K=%d input  nodes (including bias)\n", K+1);
 if (J==0) printf("No hidden nodes. Adaline network assumed.\n");
    else   printf("  J=%d hidden nodes (including bias)\n", J+1);
 printf("  I=%d output nodes\n", I);
 printf ("params.pdw file opened\n");
 printf ("Weights.pdw file opened\n");
 printf ("%s file opened\n", predict);
 printf ("%s file opened for output\n", dataNew);

 if (J==0) {
   readweights(fdw, W3, I, K);
   clearmat(DeltaW3, I, K);
 }
 else {
//  printf("About to read weights\n");
   readweights(fdw, W1, J, K);
   readweights(fdw, W2, I, J);
   clearmat(DeltaW1, J, K);
   clearmat(DeltaW2, I, J);
 }
 fclose(fdw); printf("Weights were read and file closed\n");

// Prediction phase starts here
```

```
printf("Starting Prediction Phase\n");
p = decisionWait = decisionSend = 0;      // counters
prevTimePDU = prevLengthPDU = currTimePDU = currLengthPDU = 0;
while (readpat(X, K, D, I, origX) != EOF)
//reads next pattern (PDU) from "dataNN.txt into buf"
{
  p++;
  currentType = origX[K];
  if (p>1) {
  // perfect predictions are written for the PREVIOUS PDU
     if (currentType == origX[K-1])
       // current PDU type = previous type
       fprintf(fdnew,"  W");
       // strategy for previous PDU is Wait
     else fprintf(fdnew,"  S");     // strategy is Send

     if (currentType == origX[K-1] &&
         currLengthPDU == prevLengthPDU)
         // type, time and length are similar
       fprintf(fdnew,"  W");    // strategy for previous PDU is Wait
     else fprintf(fdnew,"  S");    // strategy is Send

     if (currentType == origX[K-1] &&
         currTimePDU == prevTimePDU &&
         currLengthPDU == prevLengthPDU)
         // type, time and length are similar
       fprintf(fdnew,"  W\n");  // strategy for previous PDU is Wait
     else fprintf(fdnew,"  S\n");   // strategy is Send
  }
  fprintf(fdnew, "%s\t", buf);    // current PDU summary is written.
  prevTimePDU = currTimePDU;
  prevLengthPDU = currLengthPDU;

  forward();

  predictedType = 0;
  for (i=I; i>=1; i--)
     predictedType = predictedType*2 + (int)trunc(Y2[i]+0.5);

  if (currentType == predictedType) {  //prediction using NN
    decisionWait++;
    fprintf(fdnew,"  w");              // here prediction is wait
```

```
   }
   else {
     decisionSend++;
     fprintf(fdnew,"  s");              // here prediction is send
   }
 } //EOF on predictions file "predict.pdu"
 fprintf(fdnew,"  N  N\n");
 printf("Predictions Wait=%d, Send=%d, total=%d\n",
   decisionWait, decisionSend, p);

 fclose(fdpa);
 fclose(fdpr);
 fclose(fdnew);
 return 0;
}

// -----------------------------------------------------------------
void clearmat (double W[Dim][Dim], int N, int M){
 int i, j;
 for (i=0; i<=N; i++)
 for (j=0; j<=M; j++)
   W[i][j] = 0.0;
}

// -----------------------------------------------------------------
void readweights (FILE *fdw, double W[Dim][Dim], int N, int M) {
 int i, j, Nw, Mw;
 fscanf(fdw, ": %d %d\n", &Nw, &Mw);
 assert(Nw==N && Mw==M);
 for (i=0; i<=N; i++) {
   for (j=0; j<=M; j++) fscanf(fdw, "%lE ", &W[i][j]);
   fscanf(fdw, "\n");
 }
}

// -----------------------------------------------------------------
double g(double x) {    //sigmoid activation function
  double r;
  if (x > 50.0) r = 1.0;
  else if (x < -50.0) r = 0.0;
       else {
         r = 1.0 /(1.0 + exp(-x));
```

```
      }
//  if(r==1.0 || r==0.0)
// printf("Warning in sigmoid activation function: x=%lf\n", x);
  return r;
}

// ------------------------------------------------------------------
int readpat (double X[Dim], int K, double D[Dim], int I,
             double origX[Dim]) {
 int i, k;
 char *eof;
 origX[0] = 1.0;            // bias. origX keeps the current pattern
 for (k=1; k<K; k++)  {
   origX[k] = origX[k+1];
                    // shift left in array origX, keeping origX[0]
 }                  // to make room for next element in time series
 eof=fgets(buf, sizeof(buf), fdpr); buf[strlen(buf)-1]='\0';
 if (eof == NULL) return EOF;

 sscanf(buf,"%x%d | %*s%*s%*s%s",
        &currTimePDU, &currLengthPDU, typePDU);
 for (k=0; k<sizePDUtable; k++)
     if (strcmp(typePDU, PDUtable[k]) == 0) break;
        // searching PDU type
 if (k < 1 || k > sizePDUtable-2)
     printf("ERROR: k=%d out of range. PDU type %s unknown.\n",
             k, typePDU);
 origX[K] = k;     // puts current PDU type at end of array origX
 for (k=0; k<=K; k++)  {
   X[k] = origX[k];
 }
 normalize(X, K);
 return 1;
}

// ------------------------------------------------------------------
void normalize(double V[Dim], int N) {
 int i;
 double norm = 0.0;
 for (i=0; i<=N; i++)
   norm += fabs(V[i]);
 if (norm>0)
```

```
   for (i=0; i<=N; i++)
     V[i] /= norm;
 return;
}

// ----------------------------------------------------------------
void multAX (double A[Dim][Dim], double X[Dim], double B[Dim],
             int N, int M) {
 int i, j;
 B[0] = 0.0;
 for (i=1; i<=N; i++) {
   B[i] = 0.0;
   for (j=0; j<=M; j++)
     B[i] += A[i][j] * X[j];
 }
}

// ----------------------------------------------------------------
void forward(){
 int i,j;

 if (J==0) {
   multAX (W3, X, Net2, I, K);
   Y2[0] = 0.0;                    //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = Net2[i];              //g(Net2[i]) = Net2[i]
 }
 else {
   multAX (W1, X, Net1, J, K);
   Y1[0] = 1.0;                    //bias node
   for (j=1; j<=J; j++)
     Y1[j] = g(Net1[j]);
   multAX (W2, Y1, Net2, I, J);
   Y2[0] = 0.0;                    //this output is not used
   for (i=1; i<=I; i++)
     Y2[i] = g(Net2[i]);
 }
}

// ----------------------------------------------------------------
```

List of References

[AGS01] Christoph Ambühl, Bernd Gärtner, and Bernhard von Stengel. "A New Lower Bound for the List Update Problem in the Partial Cost Model." *Theoretical Computer Science*, **268**(1):3–16, 2001.

[APR03] Adnan Aziz, Amit Prakash, and Vijaya Ramachandran. "A Near Optimal Scheduler for Switch-Memory-Switch Routers." Technical Report TR-03-32, The University of Texas at Austin, Department of Computer Sciences, July 2003.

[BAC97] H. A. Bahr, C. W. Abate, and J. R. Collins. "Embedded Simulation for Army Ground Combat Vehicles." Technical report, STRICOM Internal Report, July 1997.

[Bah04] H. A. Bahr. *PhD dissertation: Data Bandwidth Reduction for Embedded Simulation using Concurrent Models*. PhD thesis, Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, U.S.A., December 2004.

[BCL97] M. Bassiouni, M. Chiu, M. Loper, M. Garnsey, and J. Williams. "Performance and Reliability Analysis of Relevance Filtering for Scalable Distributed Interactive Simulation." In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 7, pp. 293–331, July 1997.

[BD96] H. A. Bahr and R. F. DeMara. "A Concurrent Model Approach to Reduced Communication in Distributed Simulation." In *Proceedings of 15th Annual Workshop on Distributed Interactive Simulation*, Orlando, FL, U.S.A., September 1996.

[Ber02] Michael Berger. "Multipath Packet Switch Using Packet Bundling." In *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologies*, pp. 244–248, May 26-29, 2002.

[BM88] A. M. Baum and D. J. McMillan. "Message Passing in Parallel Real Time Continuous Systems Simulations." In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, pp. 540–549, Pasadena, CA, U.S.A., 1988.

[CD96] David P. Cebula and Paul N. DiCaprio. "Tradeoffs Involved With Separating Aggregated Data Packets Into Attribute Cluster Packets." In *Proceedings of the 14th Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 11-15, 1996.

[COM96] David B. Cavitt, C. Michael Overstreet, and Kurt J. Maly. "A Performance Analysis Model for Distributed Simulations." In *Proceedings of the 28th conference on Winter simulation*, pp. 629–636, Coronado, CA, U.S.A., December 8-11, 1996.

[Cor98] Lockheed Martin Corporation. "Advanced Distributed Simulation Technology II (ADST II) ONESAF Testbed Baseline Assessment (DO #0069) CDRL AB02 Final Report." In *Proceedings of NAECON 88*, Dayton, Ohio, May 1998.

[CST95] James O. Calvin, Joshua Seeger, Gregory D. Troxel, and Daniel J. Van Hook. "STOW Realtime Information Transfer And Networking System Architecture." In *Proceedings of the 12th Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 13-17, 1995. IST.

[CTH02] Andy Ceranowicz, Mark Torpey, Bill Helfinstine, John Evans, and Jack Hines. "Reflections on Building the Joint Experimental Federation." In *Proceedings of the 2002 I/ITSEC*, Orlando, FL, U.S.A., December 2002.

[DCV94] Paul N. DiCaprio, Carol J. Chiang, and Daniel J. Van Hook. "PICA Performance in a Lossy Communications Environment." In *11th Workshop on Standards for the Interoperability of Distributed Simulations*, volume 2, pp. 363–366, September 26-30, 1994.

[Def94] U.S. Department of Defense. "DoD Modeling and Simulation (M&S) Management." *Department of Defense Directive 5000.59*, January 4 1994.

[Deo03] Sebastian Deorowicz. *PhD Dissertation: Universal lossless Data Compression Algorithms*. PhD thesis, Silesian University of Technology, Faculty of Automatic Control, Electronics and Computer Science, Gliwice, Poland, 2003.

[DGR01] Vincent Dumas, Fabrice Guillemin, and Philippe Robert. "Effective Bandwidths in a Multiclass Priority Queueing System." Technical Report Work package 2, ALCOMFT-TR-01-178, INRIA, France, October 2001.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. "IP Header Compression.", February 1999. Internet Draft RFC 2507.

[DQ00] Sean Dorward and Sean Quinlan. "Robust Data Compression of Network Packets.", 2000.

[FL02] Jens S. Frederiksen and Kim S. Larsen. "Packet Bundling." In Martti Penttonen and Erik Meineche Schmidt, editors, *Proceedings of the Algorithm Theory - SWAT 2002: 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes in Computer Science*, pp. 328–337, Turku, Finland, July 3-5, 2002. Springer-Verlag Heidelberg.

[FLN03] Jens S. Frederiksen, Kim S. Larsen, John Noga, and Patchrawat Uthaisombut. "Dynamic TCP acknowledgment in the LogP model." *Journal of Algorithms*, **48**(2):407–428, 2003.

[For02] The U.S. Army Objective Force. "The United States Army Objective Force Operational and Organizational Plan for Maneuver Unit of Action." Pamphlet 525-3-90/ o&o, U.S. Army Training and Doctrine Command, July 22, 2002.

[Fro02] Frontlines. "JBC Initiative Delivers High-Bandwidth Collaboration Tools to Austere Locations." Technical report, http://www.microsoft.com/usa/government/MSFrontlines_summer.pdf, Summer 2002.

[Fuj95] Richard M. Fujimoto. "Parallel And Distributed Simulation." In *Proceedings of the 27th Winter Simulation Conference*, pp. 118–125, Arlington, Virginia, U.S.A., 1995. ACM Press.

[Ful96] D. Fullford. "Distributed Interactive Simulation: It's Past, Present, and Future." In *Proceedings of the 1996 Winter Simulation Conference*, Coronado, CA, U.S.A., December 8-11, 1996.

[FY94] J. Fowler and R. Yagel. "Lossless Compression of Volume Data." In *Proceedings of the 1994 Symposium on Volume Visualization*, pp. 43–50, Washington, DC, U.S.A., October 17-18, 1994.

[FZ02] Chuan Heng Foh and Moshe Zukerman. "Performance Analysis of the IEEE 802.11 MAC Protocol." In *European Wireless 2002 Conference*, Florence, Italy, February 25-28, 2002.

[GDD02] Gary Green, Michael Dolezal, Ronald F. DeMara, Avelino J. Gonzalez, Michael Georgiopoulos, and Guy Schiavone. "Embedded Simulation Research." Technical report, University of Central Florida, Electrical and Computer Engineering Department, Orlando, FL, U.S.A., March 27, 2002.

[GHP03] Ashish Goel, Monika R. Henzinger, Serge Plotkin, and Eva Tardos. "Scheduling Data Transfers in a Network and The Set Scheduling Problem." *Journal of Algorithms*, **48**(2):314–332, 2003.

[GIS03] Rajesh K. Gupta, Sandy Irani, and Sandeep Kumar Shukla. "Formal methods for Dynamic Power Management." In *International Conference on Computer Aided Design ICCAD-2003*, pp. 874–881, San Diego, La Jolla, CA, U.S.A., November 9-13, 2003.

[GM04] Fabrice Guillemin and Ravi Mazumdar. "Rate Conservation Laws for Multidimensional Processes of Bounded Variation with Applications to Priority Queueing Systems." *Methodology and Computing in Applied Probability*, **6**:136–159, 2004.

[Hew95] Hewlett-Packard Company. *WAN Link Compression on HP Routers*, May 1995. http://www.hp.com/rnd/support/manuals/pdf/comp.pdf.

[HGG00] A. E. Henninger, A. J. Gonzalez, M. Georgiopoulos, and R. F. DeMara. "Modeling Semi-Automated Forces with Neural Networks: Performance Improvement Through a Modular Approach." In *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*, Orlando, FL, U.S.A., May 16-18, 2000.

[HGG01] A. E. Henninger, A. J. Gonzalez, M. Georgiopoulos, and R. F. DeMara. "Human Performance Models for Embedded Training: A Novel Approach to Entity State Synchronization." In *Proceedings of the '01 Advanced Simulation Technology Conference–Military, Government, and Aerospace Conference (ASTC-MGA)*, Seattle, WA, U.S.A., April 22-26, 2001.

[HIL98] Sue Hoxie, Gil Irizarry, Ben Lubetsky, and Darren Wetzel. "Developments in Standards for Networked Virtual Reality." *IEEE Comput. Graph. Appl.*, **18**(2):6–9, 1998.

[HSC95] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation." *SIGCOMM*, pp. 328–341, 1995.

[Huf52] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In *Proceedings of the IRE*, volume 40, pp. 1082–1101, 1952.

[IEE85] IEEE Computer Society Press. *IEEE/ANSI Standard 8802/3. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification*, 1985.

[IEE93] IEEE Computer Society Press. *IEEE Std 1278-1993, IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulations Applications. Entity Information and Interaction*, 1993.

[IEE95a] IEEE Computer Society Press. *IEEE Std 1278.1-1995 IEEE Standard for Distributed Interactive Simulation - Application Protocols*, 1995.

[IEE95b] IEEE Computer Society Press. *IEEE Std 1278.2-1995 IEEE Standard for Distributed Interactive Simulation - Communication Services and Profiles*, 1995.

[IEE96] IEEE Computer Society Press. *IEEE Std 1278.3-1996 IEEE Recommended Practice for Distributed Interactive Simulation - Exercise Management and Feedback*, 1996.

[IEE97] IEEE Computer Society Press. *IEEE Std 802.11-1997 Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1997.

[IEE98] IEEE Computer Society Press. *IEEE Std 1278.1a-1998 IEEE Standard for Distributed Interactive Simulation - Application Protocols*, 1998.

[IEE99] IEEE Computer Society Press. *IEEE/ANSI STANDARD 8802-11. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[Ish01] Joseph A. Ishac. "Survey of Header Compression Techniques." Technical Report TM2001-211154, NASA Glenn Research Center, Cleveland, Ohio, U.S.A., September 2001.

[Kar92] Richard M. Karp. "On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?" In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92*, volume 1, pp. 416–429, Madrid, Spain, September 7-11, 1992. North-Holland.

[Kir95] Samuel A. Kirby. *"NPSNET: Software Requirements for Implementation of a Sand Table in The Virtual Environment."*. Master's thesis, Naval Postgraduate School, United States Navy, Monterey, CA 93943-5000, U.S.A., September 1995.

[KLJ00] J. Kaiser, M.A. Livani, and W. Jia. "Predictability of Message Transfer in CSMA-Networks." In *4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP2000)*, Hong Kong, China, December 2000.

[LCL99] L.A.H. Liang, Wentong Cai, Bu-Sung Lee, and S.J. Turner. "Performance Analysis of Packet Bundling Techniques in DIS." In *Proceedings of the 3rd IEEE International Workshop on Distributed Interactive Simulation and Real-Time Applications*, pp. 75–82, Greenbelt, MD, U.S.A., 1999.

[Liu02] Enjie Liu. *A Hybrid Queueing Model for Fast Broadband Networking Simulation*. PhD thesis, Queen Mary, University of London, March 2002. Dissertation Submitted for the Degree of Doctor of Philosophy, Department of Electronic Engineering.

[LS93] Randall Landry and Ioannis Stavrakakis. "A Three-Priority Queueing Policy with Application to DQDB Modeling." In *INFOCOM 1993*, volume 3, pp. 1067–1074, San Francisco, CA, U.S.A., 1993.

[Mac95] Michael R. Macedonia. *A Network Software Architecture for Large Scale Virtual Environments*. PhD thesis, Naval Postgraduate School, Monterey, CA, U.S.A., June 1995. PhD Thesis.

[MB98a] L. B. McDonald and H. A. Bahr. "Research on the Cost Effectiveness of Embedded Simulation and Embedded Training." In *Proceedings of the 98 Spring Simulation Interoperability Workshop*, Orlando, FL, U.S.A., March 1998.

[MB98b] L. B. McDonald and H. A. Bahr. "Research on the Cost Effectiveness of Embedded Simulation and Embedded Training - An Update." In *Proceedings of the 98 Fall Simulation Interoperability Workshop*, Orlando, FL, U.S.A., March 1998.

[McD88] L. B. McDonald. "Potential Benefits of Embedded Training." In *Proceedings of NAECON 88*, Dayton, Ohio, U.S.A., May 1988.

[MDF97] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. "Potential benefits of Delta-encoding and Data Compression for HTTP." In *ACM SIGCOMM'97 Conference*, pp. 181–194, September 1997.

[MDF02] Jeffrey Mogul, Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, Yaron Goland, Arthur van Hoff, and D. Hellerstein. "Delta encoding in HTTP.", January 2002. IETF Internet Draft.

[Mol94] M. Molle. "A new binary logarithmic arbitration method for Ethernet.", 1994.

[MR90] L. B. McDonald and J. C. Rullo. "Recommended Procedures for Implementing Cost-Effective Embedded Training in Operational Equipment."

In *Proceedings of the 12th Interservice/Industry Training Systems Conference*, Orlando, FL, U.S.A., November 1990.

[MWH01] B. McDonald, J. Weeks, and J. Hughes. "Development Of Computer Generated Forces For Air Force Security Forces Distributed Mission Training." In *Proceedings of the 2001 I/ITSEC*, Orlando, FL, U.S.A., November 26-29, 2001.

[MZP94] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. "NPSNET: A Network Software Architecture for Large-Scale Virtual Environments." *Presence*, **3**(4):265–287, 1994.

[Pop02] Cheryl Lynn Pope. *PhD dissertation: Scheduling and Management of Real-Time Communication in Point-To-Point Wide Area Networks.* PhD thesis, University of Adelaide, Department of Computer Science, Australia, 2002.

[PW98] Stephen G. Purdy and Roger D. Wuerfel. "A Comparison of HLA and DIS Real-Time Performance." In *Abstracts, Papers, & Presentations for 1998 Spring Simulation Interoperability Workshop*, Orlando, FL, U.S.A., March 9-13, 1998.

[PW99] Steven Phillips and Jeffrey Westbrook. "On-Line Algorithms: Competitive Analysis and Beyond." In *Algorithms and Theory of Computation Handbook,*. CRC Press, 1999.

[PW03] Erica L. Plambeck and Amy R. Ward. "Optimal Control of Assemble-to-Order Systems with Delay Guarantees." Research paper no. 1777, Stanford University, Dept. of Management Science and Engineering, Palo Alto, CA, U.S.A., March 2003.

[SH96] Joshua E. Smith and Daniel J. Van Hook. "Comparison of Consistency Protocol vs. DIS-Lite." In *Proceedings of the 14th Workshop on Standards for the Interoperability of Distributed Simulations*, pp. 875–884, Orlando, FL, U.S.A., March 1996. Institute for Simulation and Training.

[Sha48] C. E. Shannon. "A mathematical Theory of Communication." In ed. D. Slepian, editor, *Key Papers in the Development of Information Theory*, pp. 5–18, New York, 1948. IEEE Press.

[Sri96] S. Srinivasan. "Efficient Data Consistency in HLA/DIS++." In *Proceedings of the 1996 Winter Simulation Conference*, pp. 946–951, Coronado, CA, U.S.A., December 8-11, 1996.

[SZB96] Steve Stone, Mike Zyda, Don Brutzman, and John Falby. "Mobile Agents and Smart Networks for Distributed Simulation." In *Proceedings of the 14th Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 1996.

[Tay95] Darrin Taylor. "DIS-Lite & Query Protocol." In *Proceedings of the 13th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., September 1995.

[Tay96a] Darrin Taylor. "DIS-Lite & Query Protocol: Message Structures." In *Proceedings of the 14th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 11-15, 1996.

[Tay96b] Darrin Taylor. "The VR-Protocol." In *In Proceedings of the 14th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 11-15, 1996.

[Tec95] Office of Technology Assessment. "Distributed Interactive Simulation of Combat." Technical Report OTA-BP-ISS-151, U.S. Congress, Washington, DC, U.S.A., September 1995. U.S. Government Printing Office.

[TS02] A. Turpin and W. F. Smyth. "An approach to phrase selection for offline data compression." In *Proceedings of the 25th Australasian conference on Computer science*, pp. 267–273, Melbourne, Victoria, Australia, 2002. Australian Computer Society, Inc.

[US95a] Office of Technology Assessment U.S. Congress. "Distributed Interactive Simulation of Combat." Technical report, U.S. Government Printing Office OTA-BP-ISS-151, Washington, DC, U.S.A., September 1995.

[US95b] U.S. Department of Defense, Under Secretary of Defense for Acquisition and Technology. "DoD Modeling and Simulation (M&S) Master Plan." Technical Report Directive 5000.59-P, Department of Defense, October 1995.

[US98] U.S. Department of Defense, Under Secretary of Defense for Acquisition and Technology. "DoD Modeling and Simulation (M&S) Glossary." Technical Report Directive 5000.59-M, Department of Defense, January 1998.

[Var03] András Varga. "OMNeT++ Discrete Event Simulation System, Version 2.3, User Manual.", June 15 2003.

[VCM94] Daniel J. Van Hook, James O. Calvin, and Duncan C. Miller. "A Protocol Independent Compression Algorithm (PICA)." Technical report, MIT Lincoln Laboratory, April 2 1994. Project Memorandum No. 20PM-ADS-0005.

[VCN94] Daniel J. Van Hook, James O. Calvin, M. Newton, and D. Fusco. "An Approach to DIS Scalability." In *11th Workshop on Standards for the Interoperability of Distributed Simulations*, volume 2, pp. 347–356, September 26-30, 1994.

[VCR96] Daniel J. Van Hook, David P. Cebula, Steven J. Rak, Carol J. Chiang, Paul N. DiCaprio, and James O. Calvin. "Performance of STOW RITN Application Control Techniques." In *Proceedings of the 14th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, Orlando, FL, U.S.A., March 1996.

[VDG04a] Juan J. Vargas, Ronald DeMara, Avelino Gonzalez, and Michael Georgiopoulos. "Bandwidth Analysis of a Simulated Computer Network Running OTB." In *Proceedings of the Second Swedish-American Workshop on Modeling and Simulation (SAWMAS 2004)*, Cocoa Beach, FL, U.S.A., February 2004.

[VDG04b] Juan J. Vargas, Ronald F. DeMara, Michael Georgiopoulos, Avelino J. Gonzalez, and Henry Marshall. "PDU Bundling and Replication for Reduction of Distributed Simulation Communication Traffic." *JDMS: The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 2004. Submitted for publication in July 2004, currently under review.

[VGD03] Juan J. Vargas, Frank Goergen, Ronald DeMara, Avelino Gonzalez, and Michael Georgiopoulos. "Interim Report: Bandwidth and Latency Implications of Integrated Tactical and Training Communication Networks." Technical report, University of Central Florida, Department of Electrical and Computer Engineering, Orlando, FL, U.S.A., July 6, 2003.

[WAS96] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. "Removal Policies in Network Caches for World-Wide Web Documents." In *Procedings of the ACM SIGCOMM '96 Conference*, pp. 293–305, Stanford University, CA, U.S.A., August 1996.

[WH00] B. Wang and J. Hou. "Multicast Routing and Its QoS Extension: Problems, Algorithms, and Protocols." *IEEE Network*, **14**, January 2000.

[WJ98] Roger D. Wuerfel and Ronny Johnston. "Real-Time Performance of RTI Version 1.3." In *Proceedings of the 1998 Fall Simulation Interoperability Workshop*, September 14-18, 1998.

[WMS01] C. Wills, M. Mikhailov, and H. Shang. "N for the Price of 1: Bundling Web Objects for More Efficient Content Delivery." In *Proceedings of the 10th International Conference on World Wide Web, ISBN 1-58113-348-0*, pp. 257–265, Hong Kong, 2001.

[ZL77] Jacob Ziv and Abraham Lempel. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*, **23**(3):337–343, 1977.