EXata 5.1
Programmer's Guide

August 2013

SCALABLE Network Technologies, Inc.

600 Corporate Pointe, Suite 1200
Culver City, CA 90230

+1.310.338.3318 TEL
+1.310.338.7213 FAX

SCALABLE NETWORK TECHNOLOGIES

SCALABLE-NETWORKS.COM

SCALABLE Network Technologies, Inc.

600 Corporate Pointe, Suit 1200

Culver City, CA 90230

+1.310.338.3318 TEL
+1.310.338.7213 FAX

SCALABLE-NETWORKS.COM

Table of Contents

Preface

Who Should Read this Guide

The intended audience of *EXata 5.1 Programmer's Guide* are programmers who want to use the interface and programming functions in EXata for their own simulation purposes and to develop customized protocol models. It assumes you are familiar with the programming features of your operating system (Windows or Linux) and with the C++ programming environment. Additionally, this guide assumes you are familiar with network programming terminology and concepts.

How this Guide is Organized

This guide contains the following information:

- Chapter 1 introduces the different components of EXata and the protocol stack that forms the basis of the EXata architecture.
- Chapter 2 gives an overview of the EXata directory and file organization. It also provides instructions for compiling and debugging EXata, and for activating an add-on module.
- Chapter 3 gives an overview of the EXata simulation engine. It introduces discrete-event simulation and describes how protocols are modeled in EXata. It describes the types of events used in EXata and their implementation. This chapter also introduces the hierarchical architecture of the EXata simulation engine.
- Chapter 4 describes the procedures for developing and adding a custom model to EXata. It contains the following sections:

  Section 4.1 describes some tasks common to developing most models: Reading user-specified configuration parameters from an input file, Programming with message `info` fields, and Random number generation.

  Section 4.2 describes the Application Layer protocols implemented in EXata, the directories and files relevant to the Application Layer, and Application Layer data structures and APIs. This chapter gives a detailed description of the procedure to develop and add an Application Layer protocol to EXata. Two types of protocols are covered in this section: traffic-generating protocols and Application Layer routing protocols.

  Section 4.3 describes the Transport Layer protocols implemented in EXata, the directories and files relevant to the Transport Layer, and Transport Layer data structures and APIs. This section gives a detailed description of the procedure to develop and add a Transport Layer protocol to EXata.

Section 4.4 describes the Network Layer protocols implemented in EXata, the directories and files relevant to the Network Layer, and Network Layer data structures and APIs. This section gives a detailed description of the procedure to develop and add a Network Layer protocol to EXata. The following types of protocols are covered in this section: Network Layer unicast routing protocols, Network Layer multicast routing protocols, queueing protocols, and schedulers.

Section 4.5 describes the MAC Layer protocols implemented in EXata, the directories and files relevant to the MAC Layer, and MAC Layer data structures and APIs. This chapter gives a detailed description of the procedure to develop and add wired and wireless MAC protocols to EXata.

Section 4.6 describes the Physical Layer protocols implemented in EXata, the directories and files relevant to the Physical Layer, and Physical Layer data structures and APIs. This section gives a detailed description of the procedure to develop and add PHY and antenna models to EXata.

Section 4.7 describes the communication medium models implemented in EXata, the directories and files relevant to the communication medium, and communication medium data structures and APIs. This section gives a detailed description of the procedure to develop and add a communication medium model to EXata.

Section 4.8 describes the node mobility models implemented in EXata, the directories, files, and data structures relevant to node mobility models. This section gives a detailed description of the procedure to develop and add a node mobility model to EXata.

Section 4.9 describes the procedure to add trace collection to a protocol.

Section 4.10 describes the procedure to develop and add a custom add-on module to EXata.

Section 4.11 describes the procedure to enable communication between non-adjacent layers and communication among layers across nodes.

- Chapter 5 describes the GUI component of EXata and how to use it for protocol development.
- Chapter 6 describes the external interface API that allows EXata to interface with external entities such as other programs or physical devices.
- Chapter 7 describes the dynamic API that allows users and programs to dynamically modify and monitor a EXata simulation.
- Appendix A lists some coding guidelines and compatibility issues when developing EXata models for 64-bit platforms.
- Appendix B lists some coding guidelines for developing EXata models for multi-processor architectures.

EXata Document List

The following table shows the EXata Documentation Set and offers a brief description of each document.

Document	Description
EXata API Reference Guide	This guide is a supplement to *EXata Programmer's Guide* and provides detailed information on the EXata API functions and parameters. This is available in both PDF and HTML formats.
EXata Connection Manager User's Guide	This guide provides information on installing and using EXata Connection Manager.
EXata Distributed Reference Guide	This guide provides instructions for running EXata on a distributed architecture.
EXata Documentation Portfolio	The documentation portfolio combines all EXata documents in a single PDF file.

Document	Description
EXata Installation Guide	This guide provides detailed steps for installing EXata on Windows and Linux platforms.
EXata Model Libraries	This set of documents contains detailed reference information on all EXata models and includes the following protocol libraries. See *EXata Model Library Index* for an alphabetical list of all our models and a reference to which library they can be found in. Advanced Wireless Cellular Cyber Developer Federation Interfaces LTE Multimedia and Enterprise Network Management Sensor Networks UMTS Urban Propagation Wireless
EXata Product Tour	This tour provides an introduction to EXata by means of an example.
EXata Programmer's Guide	This is a guide to the EXata programming interface and functions, allowing users to develop and customize protocol models.
EXata Release Notes	This document lists the changes (added and removed features, bug fixes, etc.) made in the current version of EXata with respect to the previous version.
EXata Statistics Database User's Guide	This is a guide to the statistics database generated by EXata.
EXata User's Guide	This is a detailed guide for using *EXata* and works in combination with the *EXata Model Libraries* set of documents.

Document Conventions

EXata documents use the following conventions:

Convention	Description
Book Title	Title of a document.
`Command Input`	A command name or qualified command phrase, daemon, file, or option name.
`Command Output`	Text displayed by the computer.
Note: or **Notes:**	Information of special interest.
[ ]	In syntax definitions, square brackets indicate items that are optional.
`Code Segment`	Segment of code from EXata source files used for illustration.
`Added Code`	Example of code that the user should add to existing EXata functions and declarations to add a custom model to EXata. A vertical margin in the left column indicates new lines of code that need to be added.
Ellipses (`. . .`)	Ellipses are used to indicate lines of code from EXata source files that have been omitted from an example for the sake of brevity.

More Information

- For general information about SCALABLE, visit the company website at www.scalable-networks.com.
- For more information on EXata, please contact EXata Sales at info@scalable-networks.com or visit the EXata website at www.exata.com.
- For technical help on EXata or help on EXata documentation, please contact EXata Support at support@scalable-networks.com or visit our Support website at support.scalable-networks.com.

1 Introduction

EXata provides a comprehensive set of tools with all the components for custom network modeling and simulation projects. EXata's unparalleled speed, scalability, and fidelity make it easy for modelers to optimize existing networks through quick model setup and in-depth analysis tools. Models in source form provide developers with a solid library on which to build and experiment with new network functionality. The end result is accurate prediction of network performance for a diverse set of application requirements and uses. From wired LANs and WANs, to cellular, satellite, WLANs and mobile ad hoc networks, EXata's library is extensive. Because of its efficient kernel, EXata models large scale networks with heavy traffic and mobility in reasonable simulation times.

This chapter gives a brief introduction to the different components of EXata, and introduces the protocol stack that forms the basis of EXata architecture.

1.1 EXata Components

EXata has several core components, as well as various add-on components. This section provides a brief description of the core components of EXata. Detailed descriptions, functions, and usage instructions for each of the EXata components are available in *EXata User's Guide*.

EXata Simulator

EXata Simulator is a state-of-the-art simulator for large, heterogeneous networks and the distributed applications that execute on those networks. EXata Simulator is an extremely scalable simulation engine, accommodating high-fidelity models of networks of tens of thousands of nodes. EXata makes good use of computational resources and models large-scale networks with heavy traffic and mobility, in reasonable simulation times.

EXata Simulator has the following attractive features:

- Fast model set up with a powerful Graphical User Interface (GUI) for custom code development and reporting options
- Instant playback of simulation results to minimize unnecessary model executions
- Fast simulation results for thorough exploration of model parameters
- Scalable up to tens of thousands of nodes
- Real-time simulation for man-in-the-loop and hardware-in-the-loop models
- Multi-platform support

EXata Architect

EXata Architect is a graphical tool that provides an intuitive model set up and execution capability. Architect has two modes: Design mode and Visualize mode.

In Design mode, Architect is used to create and design experiments. Architect enables a user to define the geographical distribution, physical connections and the functional parameters of the network nodes, all using intuitive click and drag tools, and to define network layer protocols and traffic characteristics for each node.

In Visualize mode, Architect is used to execute and animate experiments created in the Design mode. Using Architect, a user can watch traffic flow through the network and create dynamic graphs of critical performance metrics as a simulation is running.

EXata Analyzer

EXata Analyzer statistical graphing tool that displays network statistics generated from a EXata experiment. Using the Analyzer, a user can view statistics as they are being generated, as well as compare results from different experiments.

1.2 EXata Protocol Stack

EXata uses a layered architecture similar to that of the TCP/IP network protocol stack. Within that architecture, data moves between adjacent layers. EXata's protocol stack consists of, from top to bottom, the Application, Transport, Network, Link (MAC) and Physical Layers.

Adjacent layers in the protocol stack communicate via well-defined APIs, and generally, layer communication occurs only between adjacent layers. For example, Transport Layer protocols can get and pass data to and from the Application and Network Layer protocols, but cannot do so with the Link (MAC) Layer protocols or the Physical Layer protocols. This rule concerning communication only between adjacent layers may be circumvented by the programmer, as explained in Section 4.11.

Figure 1-1 depicts the EXata protocol stack and the general functionality of each layer.

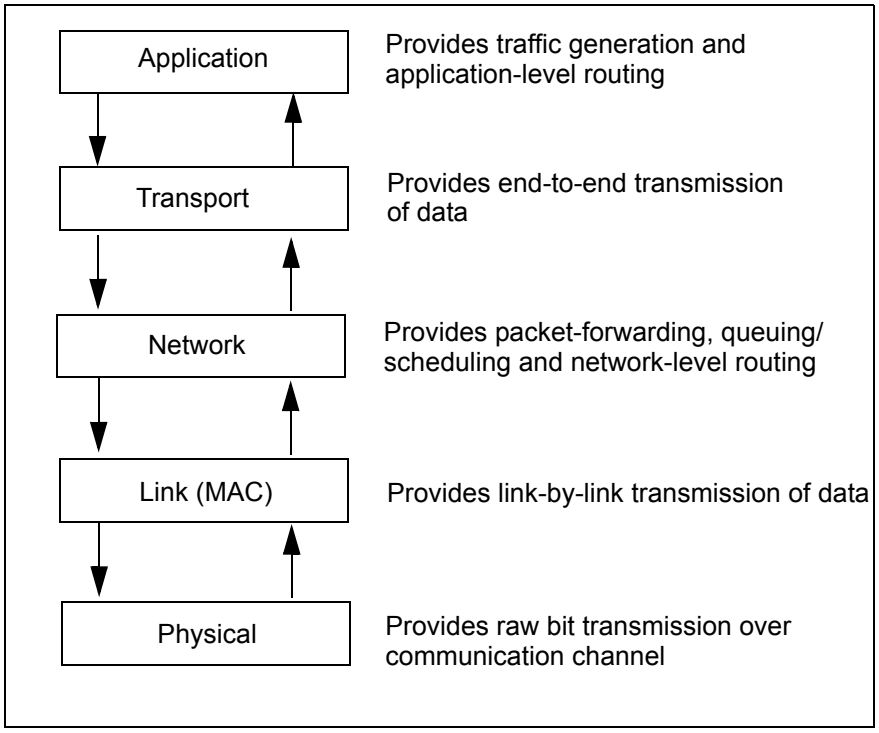


FIGURE 1-1. EXata Protocol Stack

1.2.1 Application Layer

The Application Layer is responsible for traffic generation and application level routing. Protocols written at the Application Layer rely on the Transport Layer to deliver application-level data from the source to the destination. Thus, Application Layer protocols pass data down to the Transport Layer at the source node, and receive data from the Transport Layer at the destination node. Examples of traffic- generating Application Layer protocols implemented in EXata are Constant Bit Rate (CBR), FTP, and Telnet. Examples of Application Layer routing protocols implemented in EXata are RIP, Bellman-Ford, and BGP.

Section 4.2 provides implementation details of Application Layer protocols in EXata and describes how to develop a custom Application Layer protocol.

1.2.2 Transport Layer

The Transport Layer provides end-to-end data transmission services to the Application Layer. Protocols written at the Transport Layer receive data from the Application Layer and rely on the Network Layer for data forwarding at the source node, and receive data from the Network Layer and pass data to the Application Layer at the destination node. Examples of Transport Layer protocols include UDP, TCP and RSVP-TE.

Section 4.3 provides implementation details of Transport Layer protocols in EXata and describes how to develop a custom Transport Layer protocol.

1.2.3 Network Layer

The Network Layer is responsible for data forwarding and queuing/scheduling. The Internet Protocol (IP) resides at this layer and is responsible for packet forwarding. At the source node, the Network Layer receives data from the Transport Layer and relies on the Link (MAC) Layer for link-by-link data delivery. At the destination node, the Network Layer receives data from the Link (MAC) Layer and passes the data up to the Transport Layer.

The Network Layer also implements certain types of routing protocols. Examples of Network Layer routing protocols implemented in EXata are AODV, DSR, OSPF, and DVMRP. Examples of queuing/scheduling protocols implemented in EXata are FIFO, RED, RIO, WFQ, and WRR.

Section 4.4 provides implementation details of Network Layer protocols in EXata and describes how to develop a custom Network Layer protocol. The following types of protocols are covered in this section: Network Layer unicast routing protocols, Network Layer multicast routing protocols, queuing protocols, and schedulers.

1.2.4 Link (MAC) Layer

The Link (MAC) Layer provides link-by-link transmission. At the sending side, the Link (MAC) Layer receives data from the Network Layer and passes the data to the Physical Layer for transmission over the wired or wireless channel. At the receiving side, the Link (MAC) Layer receives data from the Physical Layer and forwards the data up to the Network Layer. Examples of protocols at the Link (MAC) Layer implemented in EXata are point-to-point, IEEE 802.3, IEEE 802.11, and CSMA.

Section 4.5 provides implementation details of MAC Layer protocols in EXata and describes how to develop a custom MAC Layer protocol. Procedures for both wired and wireless MAC protocols are covered in this section.

1.2.5 Physical Layer

The Physical Layer is responsible for transmitting and receiving raw bits from the wired and wireless channel. At the source node, the Physical Layer receives data from the Link (MAC) Layer and sends the data to the Physical Layer of the destination node. At the destination node, the Physical Layer receives data from the Physical Layer of the source node and passes the data to the Link (MAC) Layer.

Note: For wired networks, the Physical Layer code is incorporated into the Link (MAC) Layer.

Examples of Physical Layer protocols implemented in EXata are wired point-to-point links, IEEE 802.3, and IEEE 802.11.

Section 4.6 provides implementation details of Physical Layer protocols in EXata and describes how to develop a custom Physical Layer protocol.

1.2.6 Communication Medium

The communication medium transmits signals between nodes. It interfaces with the Physical Layer entities at the nodes. A wireless communication medium model in EXata simulates the propagation of signals between nodes, taking into account both propagation delays and signal attenuation due to path loss, fading, and shadowing.

In EXata, a communication medium model has three components: a path loss model, a fading model, and a shadowing model. Path loss models in EXata include free space, two ray, and Irregular Terrain Model (ITM). EXata implements the Ricean fading model. Rayleigh fading is a special case of Ricean fading. EXata provides models for two shadowing models: constant and lognormal.

Section 4.7 provides implementation details of communication medium models in EXata and describes how to develop a custom communication medium model.

1.2.7 Node Mobility

In EXata, mobility models work together with node placement models and terrain models to simulate the mobility behavior of nodes. Node mobility models in EXata include random waypoint, group mobility, pedestrian mobility, and file-base mobility.

Section 4.8 gives a detailed description of how to add a mobility model to EXata.

2 EXata File Organization, Compilation, and Debugging

In this chapter, we describe the file organization in EXata and how to compile, install addons, and debug EXata.

Section 2.1 describes the directory structure of EXata.

Section 2.2 describes how to compile EXata on Windows platforms.

Section 2.3 describes how to compile EXata on Linux platforms. (For compiling EXata on distributed platforms, refer to *EXata Distributed Reference Guide*.)

Section 2.4 describes how to activate and deactivate EXata addons.

Section 2.5 describes advanced options for compiling EXata.

Section 2.6 describes how to debug EXata.

2.1 File Organization

EXata distribution files are grouped into several subdirectories. This allows users to quickly find source code, binary object files, configuration files, documentation, or samples. Table 2-1 lists the subdirectories and their contents.

Note: In this document, EXATA_HOME refers to the EXata installation directory. This is stored as an environment variable for Windows and Linux platforms.

TABLE 2-1. Default EXata Subdirectories

Subdirectory	Description
EXATA_HOME/addons	Components developed as custom add-on modules
EXATA_HOME/bin	Executable and other runtime files, such as DLLs
EXATA_HOME/contributed	Files related to models contributed by third parties
EXATA_HOME/data	Data files for the Wireless Model Library, including antenna configurations, modulation schemes, and sample terrain files.
EXATA_HOME/documentation	Documentation (User's Guide, Release Notes, etc.)
EXATA_HOME/gui	Graphical components, including icons, and GUI configuration files
EXATA_HOME/include	EXata kernel header files
EXATA_HOME/installers	Installers for supplemental third party software
EXATA_HOME/interfaces	Code to interface EXata with third party tools or external networks, such as HLA and DIS
EXATA_HOME/kernel	EXata kernel objects used in the build process
EXATA_HOME/lib	Third party software libraries used in the build process
EXATA_HOME/libraries	Source code for models in EXata model libraries, such as Developer, Wireless, and Multimedia & Enterprise.
EXATA_HOME/license_dir	License files and license libraries required for the build process
EXATA_HOME/main	Kernel source files and Makefiles
EXATA_HOME/scenarios	Sample scenarios

2.2 Compiling EXata on Windows

This section describes how to compile EXata on Windows platforms. Section 2.2.1 lists the supported C++ compilers. Section 2.2.2 describes the precompiled executable files included in the EXata distribution. Section 2.2.3 gives detailed instructions for compiling EXata.

2.2.1 C++ Compiler

One of the C++ compilers listed in Table 2-2 is required to compile EXata. For convenience, this guide will refer to the compilers by their abbreviations.

TABLE 2-2. C++ Compilers for Windows

C++ Compiler	Abbreviation
Microsoft Visual Studio 2008	VC9
Microsoft Visual C++ 2008 Express Edition	VC9 Express
Microsoft Visual Studio 2010	VC10
Microsoft Visual C++ 2010 Express Edition	VC10 Express

Note: Microsoft Visual C++ 2008 Express Edition and Microsoft Visual C++ 2010 Express Edition are available as free downloads.

To use Microsoft Visual C++ 2008 Express Edition on a 64-bit platform, Windows /.NET 3.5 Platform SDK must also be installed. Windows /.NET 3.5 Platform SDK is also available as a free download.

To use Microsoft Visual C++ 2010 Express Edition on a 64-bit platform, Windows SDK 7.1 must also be installed. Windows SDK 7.1 is also available as a free download.

Go to the Microsoft website to download these software packages.

2.2.2 Executable Files

For Windows platforms, the EXata distribution includes the following executable files:

- exata-precompiled-32bit.exe: This is a 32-bit executable that can run on both 32-bit and 64-bit platforms.
- exata.exe: This is a copy of exata-precompiled-32bit.exe.

  Note that exata.exe is overwritten every time you recompile EXata. If you recompile EXata but want to use the pre-built executable, then copy the file exata-precompiled-32bit.exe to exata.exe.

  Note: If you copy exata-precompiled-32bit to exata.exe, you must also copy libexpat.dll and pthreadVC2.dll from EXATA_HOME/lib/windows to EXATA_HOME/bin.

These executable files have been compiled with all model libraries that do not require third-party software (see Section 2.4). EXata does not need to be recompiled in order to use the models in these libraries. However, EXata will need to be recompiled if the source code is modified or certain addons are included (see Section 2.4).

Your license file will enable the model libraries that are part of the base EXata distribution (see Section 2.4) and any additional model libraries purchased by you.

2.2.3 Compiling EXata

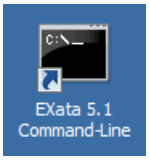
To compile EXata from the command line, follow the instructions given in Section 2.2.3.1. To compile EXata from the VC9 or VC9 Express IDE, follow the instructions given in Section 2.2.3.2. To compile EXata from the VC10 or VC10 Express IDE, follow the instructions given in Section 2.2.3.3.

2.2.3.1 Compiling from Command Line

To compile EXata from the command line, perform the steps listed below.

Note: EXata is a 32-bit application which can run on both 32-bit and 64-bit platforms. The following steps will create a 32-bit executable on both 32-bit and 64-bit platforms.

1. If EXata desktop shortcuts are installed, open the EXata command window by double-clicking on the following icon on the desktop.



Verify that the environment variables are properly set by typing the following command: `cl`.

The following output verifies that the configuration is correct:

```
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42
for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

(The compiler version in the above output may differ depending on which version of C++ is installed.)

If the configuration is correct, then skip step 2 and go to step 3.

If a “file not found” error is displayed, then go to step 2.

2. If EXata desktop shortcuts are not installed or a “file not found” error was displayed in step 1, then open a command window using one of the commands listed below.
   - To create a 32-bit executable using VC9 on 32-bit or 64-bit platforms, use the following command:

     Start > All Programs > Microsoft Visual Studio 2008 > Visual Studio Tools > Visual Studio 2008 Command Prompt

   - To create a 32-bit executable using VC9 Express on 32-bit or 64-bit platforms, use the following command:

     Start > All Programs > Microsoft Visual C++ 2008 Express Edition > Visual Studio Tools > Visual Studio 2008 Command Prompt

- To create a 32-bit executable using VC10 on 32-bit or 64-bit platforms, use the following command:

  Start > All Programs > Microsoft Visual Studio 2010 > Visual Studio Tools > Visual Studio 2010 Command Prompt

- To create a 32-bit executable using VC10 Express on 32-bit or 64-bit platforms, use the following command:

  Start > All Programs > Microsoft Visual C++ 2010 Express Edition > Visual Studio Tools > Visual Studio 2010 Command Prompt

3. Go to EXATA_HOME/main directory.
4. While installing EXata, the installer creates a makefile (called Makefile) for the 32-bit executable (for 32-bit platforms) or the 64-bit executable (for 64-bit platforms) for Microsoft Visual Studio 2010 in the EXATA_HOME/main directory.

   EXATA_HOME/main also includes makefiles for different combinations of compilers and platforms (see Table 2-4). If Makefile does not exist in EXATA_HOME/main directory or if you want to use a different makefile, then make a copy of the appropriate makefile.

TABLE 2-3. Windows Makefiles

Compiler	Makefile for 32-bit Executable (for 32-bit and 64-bit Platforms)
VC9	Makefile-windows-vc9
VC9 Express	Makefile-windows-vc9
VC10	Makefile-windows-vc10
VC10 Express	Makefile-windows-vc10

   For example, for VC9 on a 32-bit platform, use the following command to make a copy of the makefile:

```
copy Makefile-windows-vc9 Makefile
```

5. Compile EXata by using the following command (it takes several minutes for EXata to compile):

```
nmake
```

   This creates the EXata executable in the EXATA_HOME/bin directory. In Windows, the executable is called exata.exe.
6. To recompile EXata, run nmake again. However, it is sometimes useful to delete all object files before recompiling. Use the following commands to remove all object (.obj) files and recompile:

```
nmake clean
nmake
```

2.2.3.2 Compiling from Visual Studio 2008 or Visual C++ 2008 Express Edition IDE

This section describes how to compile EXata using the Microsoft Visual Studio 2008 IDE. The Microsoft Visual C++ 2008 Express Edition IDE can be used in a similar way to compile EXata.

Configuring Visual Studio 2008 or Visual C++ 2008 Express Edition IDE

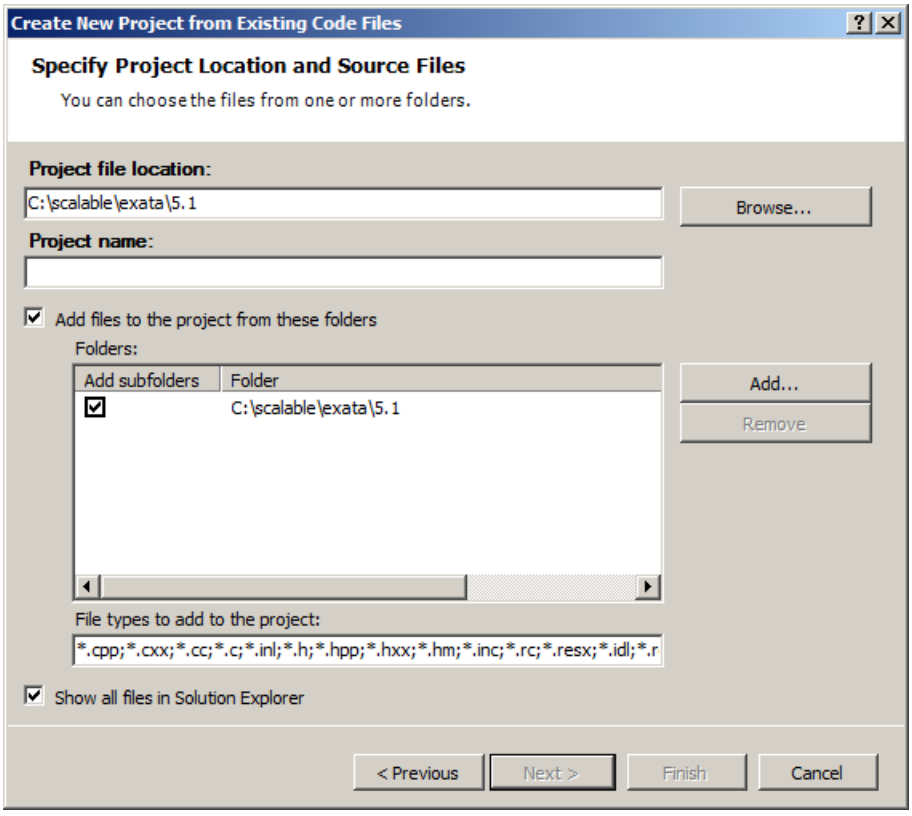
If Microsoft Visual Studio 2008 or Microsoft Visual C++ 2008 Express Edition IDE is used, then the IDE must be configured before EXata can be compiled. This configuration needs to be done only once. Perform the following steps to configure the IDE.

1. Using a text editor, create a file named *Makefile* in the EXATA_HOME directory. This file contains commands to redirect to a makefile in EXATA_HOME/main. (The indentations in the following file are tabs, not spaces.)

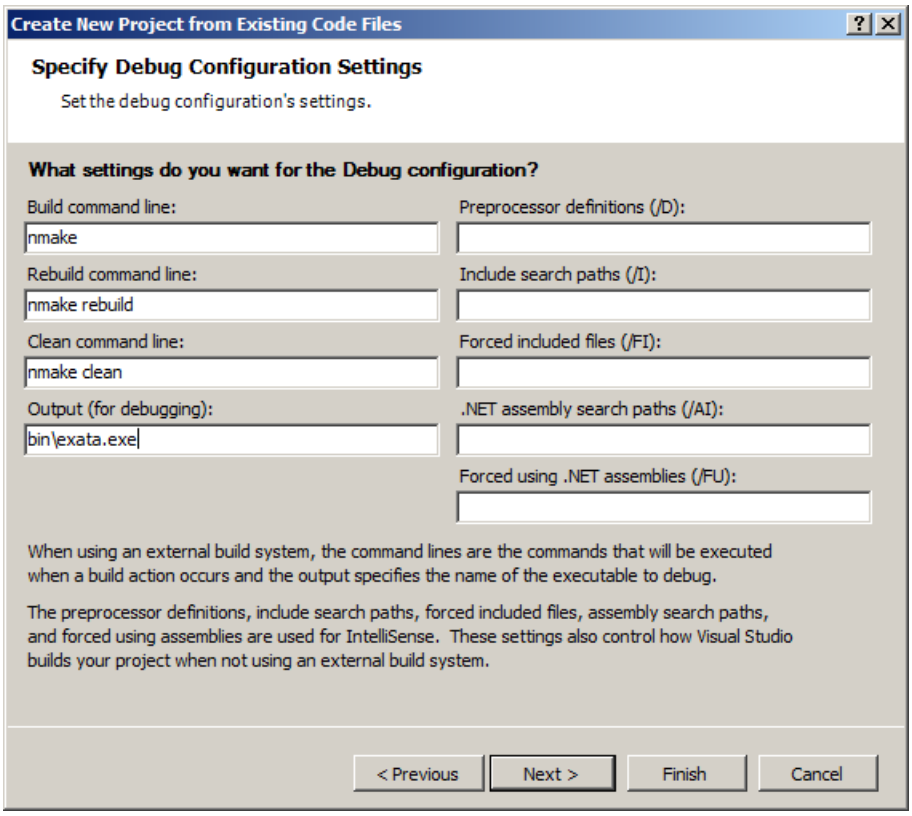
```
all:
     cd main
     nmake -f Makefile-windows-vc9

rebuild: clean
     cd main
     nmake -f Makefile-windows-vc9

clean:
     cd main
     nmake -f Makefile-windows-vc9 clean
```

2. Open Microsoft Visual Studio 2008 or Microsoft Visual C++ 2008 Express Edition.

3. Select **File > New > Project From Existing Code**.



4. Enter the following project information:
   - **Project file location**: *C:\scalable\exata\5.1*
   - **Project name**: *exata*
   - **Folder**: *C:\scalable\exata\5.1*
5. Click **Next** to continue.
6. In the next window, check the following option box:
   - **How do you want to build the project?**: *Use external build system*
7. Click **Next** to continue.
8. Set the following debug configuration settings:
   - **Build command line**: *nmake*
   - **Rebuild command line**: *nmake rebuild*
   - **Clean command line**: *nmake clean*
   - **Output (for debugging)**: *bin\exata.exe*



9. Click **Finish** to accept the settings.

Compiling from Visual Studio 2008 or Visual C++ 2008 Express Edition IDE

Once Microsoft Visual Studio 2008 or Microsoft Visual C++ 2008 Express Edition IDE has been configured, it can be used as follows:

- Select **Build > Build Solution** to build EXata.
- Select **Build > Rebuild Solution** to clean the object files and rebuild EXata.
- Select **Build > Clean Solution** to clean the object files.

2.2.3.3 Compiling from Visual Studio 2010 or Visual C++ 2010 Express Edition IDE

The instructions for compiling EXata using the Microsoft Visual Studio 2010 or Microsoft Visual C++ 2010 Express Edition IDE are virtually identical to the 2008 versions (see Section 2.2.3.2), except that for the 2010 versions you will need to replace `vc9` with `vc10`, e.g., `nmake -f Makefile-windows-vc10`.

2.3 Compiling EXata on Linux

This section describes how to compile EXata on Linux platforms. Section 2.3.1 lists the third-party software (expat development library and supported C++ compilers) required to compile EXata. Section 2.3.2 describes the precompiled executable files included in the EXata distribution. Section 2.3.3 gives detailed instructions for compiling EXata.

2.3.1 Third party Software

The expat development library and a C/C++ compiler are required to recompile EXata. This section describes how to install the required software on a Linux system.

2.3.1.1 Expat Development Library

The expat development library is needed to compile EXata on Linux systems.

Install the expat development library from the Linux installation media or download site. Consult your system administrator for help with installing the expat development library.

2.3.1.2 C/C++ Compiler

To recompile EXata source code or custom addons, a C compiler (gcc) and C++ compiler (g++) are required. Install the version of gcc appropriate for your system.

The version of gcc depends on the glibc version of the Linux distribution. To determine your version of glibc, do the following:

- For Debian-based distributions (such as Ubuntu), run the following command:

```
dpkg -s libc6
```

- For the other Linux distributions, run the following command:

```
rpm -q glibc
```

See Table 2-4 to determine the gcc version required for your system.

TABLE 2-4. C++ Compilers for Linux

Distribution	glibc Version	gcc Version
CentOS 5.9	2.5	4.1
Red Hat Enterprise Linux 5.9	2.5	4.1
Ubuntu 12.04 LTS	2.15	4.6

Install the right version of gcc from the Linux installation media or download site. Consult your system administrator for help with installing gcc.

Note: To check the version of gcc installed on your system, type the following command in a command window: `gcc -v`.

Most gcc installations include g++. If g++ is not included, then install the version of g++ compatible with the version of gcc installed. Consult your system administrator for help with installing g++.

Note: To enquire about EXata support on operating systems or compilers not listed in Table 2-4, contact sales@scalable-networks.com.

2.3.2 Executable Files

For Linux platforms, the EXata distribution includes the following executable files:

- exata-precompiled-32bit (included only for 32-bit platforms): This is a 32-bit executable that can run on 32-bit platforms.
- exata-precompiled-64bit (included only for 64-bit platforms): This is a 64-bit executable that can run on 64-bit platforms.
- exata: This is a copy of exata-precompiled-32bit for 32-bit platforms and a copy of exata-precompiled-64bit for 64-bit platforms.

  Note that the file exata is overwritten every time you recompile EXata. If you recompile EXata but want to use the pre-built executable, then copy the file exata-precompiled-32bit or exata-precompiled-64bit to exata.

Note: The executable files will run only on the machine on which EXata is installed. To use EXata on a different machine, it must be installed on that machine.

These executable files have been compiled with all model libraries that do not require third-party software (see Section 2.4). EXata does not need to be recompiled in order to use the models in these libraries. However, EXata will need to be recompiled if the source code is modified or certain addons are included (see Section 2.4).

Your license file will enable the model libraries that are part of the base EXata distribution (see Section 2.4) and any additional model libraries purchased by you.

2.3.3 Compiling EXata

To compile EXata on a Linux system, perform the following steps:

1. Open a command window.
2. Go to EXATA_HOME/main folder.
3. EXATA_HOME/main includes makefiles for different combinations of glibc and gcc versions. Make a copy of the makefile appropriate for the glibc and gcc versions installed on your system (see Table 2-5).

Notes:
1. To check the version of gcc installed on your system, type the following command in a command window: `gcc -v`.
2. To check the version of glibc installed on your system, type one the following commands in a command window:
   - For Debian-based distributions (such as Ubuntu): `dpkg -s libc6`
   - For the other Linux distributions: `rpm -q glibc`
3. If there is no makefile listed for the glibc and gcc versions installed on your system, try the makefile that is closest to your versions. However, the makefile may not work for your system. If you need help, contact support@scalable-networks.com.

TABLE 2-5. Linux Makefiles

Distribution	glibc Version	gcc Version	Makefile for 32-bit Executable (for 32-bit and 64-bit Platforms)	Makefile for 64-bit Executable (for 64-bit Platforms)
CentOS 5.9 Red Hat Enterprise Linux 5.9	2.5	4.1	Makefile-linux-glibc-2.5-gcc-4.1	Makefile-linux-x86_64-glibc-2.5-gcc-4.1
Ubuntu 12.04	2.15	4.6	Makefile-linux-glibc-2.15-gcc-4.6	Makefile-linux-x86_64-glibc-2.15-gcc-4.6

For example, for Red Hat Enterprise Linux 5.9 and other Linux distributions with glibc 2.5 and gcc 4.1 on a 32-bit platform, use the following command to make a copy of the makefile:

```
cp Makefile-linux-glibc-2.5-gcc-4.1 Makefile
```

4. Compile EXata by using the following command (it takes several minutes for EXata to compile):

```
make
```

This creates the EXata executable in the EXATA_HOME/bin directory. For Linux systems, the executable is called exata.

To recompile EXata, run **make** again. However, it is sometimes useful to delete all object files before recompiling. Use the following commands to remove all object (.o) files and recompile:

```
make clean
make
```

2.4 Activating and Deactivating Addons

Addons are components of EXata which provide enhanced features and functionality. Some of these are included with the EXata distribution, whereas others are distributed separately. Addons may contain additional source code, pre-compiled library files, and third party utilities. Addons can be activated or deactivated independently.

EXata addons fall into the following classes:

- Libraries: Protocol (model) libraries sold with EXata
- Interfaces: External interfaces requiring third party software
- Custom addons: Special purpose addon modules and user-developed addons
- Contributed models: Models developed and provided by Scalable Network Technologies customers for distribution

Model Libraries Precompiled with EXata

The source code, scenarios, and documentation for the following model libraries are included in the EXata distribution. In addition, the precompiled executable files included in the distribution (see Section 2.2.2,

and Section 2.3.2) have been compiled with the following libraries that are part of the base EXata distribution:

- Developer (including STK interface)
- Multimedia and Enterprise
- Network Management
- Wireless

and the following addon libraries:

- Advanced Wireless
- Cellular
- Cyber
- Federation Interfaces
- LTE
- Sensor Networks
- UMTS
- Urban Propagation

To deactivate or reactivate any of the addon libraries, see Section 2.4.1 (for Windows) or Section 2.4.2 (for Linux).

Model Libraries and Addons Not Included in EXata Distribution

Any model library or addon not included in the EXata distribution will need to be downloaded separately and EXata will need to be recompiled in order to use the library or addon. For instructions for downloading the model library or addon and any additional requirements, refer to the library or addon documentation. For details of compiling EXata with the model library or addon, see Section 2.4.1 (for Windows) or Section 2.4.2 (for Linux).

Section 2.4.1 describes how to activate and deactivate addons on Windows systems. Section 2.4.2 describes how to activate and deactivate addons on Linux systems.

Note: In the following sections, we use the Cellular Model Library addon as an example. Users should modify the addon name in the following instructions to match the name of the addon they want to activate or deactivate.

2.4.1 Activating and Deactivating Addons on Windows

To activate or adjective an EXata addon on Windows, perform the following steps:

1. Open the file EXATA_HOME/main/Makefile-addons-windows with a text editor.
2. To activate the addon, locate and uncomment the include statement for the addon makefile.

   For the Cellular Model Library, change the line

   ```
   #include ../libraries/cellular/Makefile-windows
   ```

   to

   ```
   include ../libraries/cellular/Makefile-windows
   ```

   To deactivate the addon, comment out the include statement for the addon makefile.

   If you want to activate an addon for which there is no include statement, then add an include statement for the addon makefile similar to the one for the Cellular Model Library.
3. Recompile EXata, as described in Section 2.2.3.

Note: Delete all object (.obj) files before recompiling by using the **`nmake clean`** command.

2.4.2 Activating Addons on Linux

To activate or deactivate an EXata addon on Linux systems, perform the following steps:

1. Open the file EXATA_HOME/main/Makefile-addons-unix with a text editor.
2. To activate the addon, locate and uncomment the include statement for the addon makefile.

   For the Cellular Model Library, change the line

   ```
   #include ../libraries/cellular/Makefile-unix
   ```

   to

   ```
   include ../libraries/cellular/Makefile-unix
   ```

   To deactivate the addon, comment out the include statement for the addon makefile.

   If you want to activate an addon for which there is no include statement, then add an include statement for the addon makefile similar to the one for the Cellular Model Library.
3. Recompile EXata, as described Section 2.3.3.

Note: Delete all object (.o) files before recompiling by using the **`make clean`** command.

2.5 Advanced Compilation Options

The folder EXATA_HOME/main contains several Makefiles which are used for building EXata. The Makefiles have been structured in a platform-independent way so that minimal changes are required to modify the build process. The Makefile organization is shown in Table 2-6.

TABLE 2-6. Makefile Organization

File	Description
Makefile-addons-unix	Makefile to include different addons for Linux platforms.
Makefile-addons-windows	Makefile to include different addons for Windows platforms.
Makefile-common	Platform-neutral makefile. Defines the master list of source files, include directories, and kernel object files.
Makefile-unix-common	Rules for Linux platforms. Included in the Linux makefiles.
Makefile-unix-exata-first-target	Contains target 'all' for Linux platforms. Included in the Linux makefiles.
Makefile-windows-common	Rules for Windows platforms. Included in the Windows makefiles (see Table 2-4).
Makefile-windows-first-target	Contains target 'all' for Windows platforms.Included in the Windows makefiles (see Table 2-4).
Makefile-windows-targets	Rules for Windows platforms defined last so they can use macro values such as $(OPT) and $(DEBUG). Included in the Windows makefiles (see Table 2-4).

The common files included in Windows and Linux Makefiles are different because the Visual C++ development environment differs from UNIX compilers and they have different sets of commands. However, the general organization of the files is similar. The Makefile-[platform] files are used to make changes to the OPT, DEBUG and FLAGS macros as appropriate for that platform. Other modifications should be made to the files that it includes. For instance, the user can add or modify compiler flags in Makefile-unix-common and Makefile-windows-common files.

Example

The following segment from EXATA_HOME/main/Makefile-windows-common shows how to add a compiler option:

```
CXXFLAGS = \
/GX /MT /nologo \
$(INCLUDE_DIRS) \
$(FLAGS) \
$(DEBUG) \
$(OPT) \
$(ADDON_OPTIONS) \
-DTEST_FLAG
```

To compile customized source files into EXata, follow the instructions in Section 4.10 for creating an Addon, Library or Interface.

2.6 Debugging EXata

This section describes how to debug EXata on Windows systems (see Section 2.6.1) and on Linux systems (see Section 2.6.2)

2.6.1 Debugging on Windows

To run the debugger, EXata must be compiled with the debug option. (By default, EXata is compiled with the optimization option for runtime efficiency.) This section describes how to compile EXata with the debug option and how to debug EXata in Microsoft Visual C++ 2008 or 2010 Express Editions (debugging in Microsoft Visual Studio 2008 or 2010 is similar).

Compiling EXata with Debug Option

Perform the following steps to recompile EXata with the debug option:

1. Open the command window for your compiler as described in section Section 2.2.3.1. Go to EXATA_HOME/main.
2. Copy the makefile for your compiler (see Section 2.2.3) to Makefile.
3. Edit Makefile as follows:
   - Enable the DEBUG line by removing the '#' character so it is displayed as:

     ```
     DEBUG = /Zi
     ```

   - Disable the OPT line by inserting a '#' character at the beginning of the line so it is displayed as:

     ```
     # OPT = /Ox /Ob2
     ```

4. Recompile EXata by typing the following commands:

   ```
   nmake clean
   nmake
   ```

Debugging in Visual C++ 2008 or 2010 Express Editions

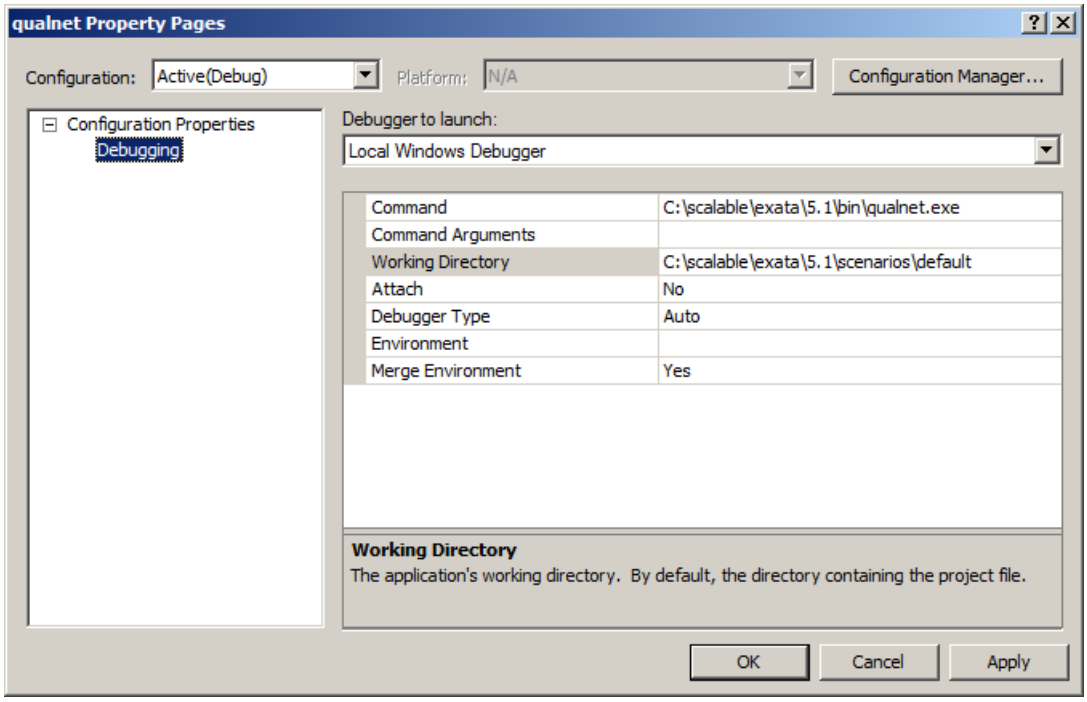
To debug EXata, perform the steps described below if you are using Visual C++ 2008 or 2010 Express Edition. (Steps to debug in Microsoft Visual Studio 2008 or 2010 are similar).

1. Start Microsoft Visual C++ 2008 or 2010 Express Edition.
2. Select **File > Open > Project/Solution** and select the executable file (exata.exe).
3. Select **Project > Properties**.
4. In the left panel, select **Debugging** under **Configuration Properties**.

   Set **Command** to the path where the executable is located.

   Set **Command Arguments** to the name of the scenario configuration (.config) file to be debugged.

   Set **Working Directory** to the directory where the scenario configuration file is located.



5. Set the breakpoints as desired and debug using the commands listed in the **Debug** menu.

Notes: **1.** When starting the program, several messages in one of the debugger status windows may be displayed that look like the following:

```
Loaded 'C:\WINNT\SYSTEM32\COMCTL32.DLL', no matching
symbolic information found.
```

This is normal.

2. If the following error occurs, the configuration file specified in the **Project > Properties** window is not in the working directory specified:

```
The thread 0x498 has exited with code 3 (0x3).
The program 'C:\scalable\exata\5.1\bin\exata.exe' has
exited with code 3 (0x3).
```

2.6.2 Debugging on Linux Systems

To run the debugger on Linux systems, EXata must be compiled with the debug option. The Linux Makefiles already contain the -g compiler option by default to include debugging information in the EXata executable.

Compiling EXata with Debug Option

By default, the optimization option of the compiler is enabled in the Linux Makefiles. When the optimization option is enabled, the compiler may optimize the program for better performance. However, for better source level debugging, the optimization option of the compiler should be disabled and the debug option should be enabled.

Perform following steps to recompile EXata with the debug option enabled:

1. Go to EXATA_HOME/main. Copy the makefile for your compiler (see Section 2.3.3) to Makefile.
2. Edit Makefile as follows:
   - Enable the DEBUG line by removing the '#' character so it is displayed as:

     ```
     DEBUG = -g
     ```

   - Disable the OPT line by inserting a '#' character at the beginning of the line so it is displayed as:

     ```
     # OPT = -O3
     ```

3. Recompile EXata by typing the following commands:

```
make clean
make
```

Running the Debugger

EXata can be run from within debug tools such as gdb or dbx. Here, we use gdb as an example.

To run gdb, perform the following steps:

1. Open a command window. Go to the directory where the scenario to be debugged is located.
2. Load the EXata executable into gdb by typing the following command (this assumes that EXata is installed in /home/username/scalable/exata/5.1):

```
gdb /home/username/scalable/exata/5.1/bin/exata
```

3. From within the gdb environment, run your scenario in gdb by typing the following command (assuming the scenario configuration file is myscenario.config):

```
run myscenario.config
```

4. To exit gdb, type following command in gdb:

```
quit
```

Refer to gdb user manual for more information on how to debug a program in gdb.

3 Simulator Basics

In this chapter, we discuss the basics of EXata Simulator. Section 3.1 provides an overview of discrete-event simulation, and Section 3.2 describes how protocols are modeled in EXata. Section 3.3 provides implementation details of discrete-event simulation in EXata, while Section 3.4 describes the architecture of EXata Simulator.

3.1 Overview of Discrete-event Simulation

EXata is a discrete-event simulator. In discrete-event simulation, a system is modeled as it evolves over time by a representation in which the system state changes instantaneously when an *event* occurs, where an event is defined as an instantaneous occurrence that causes the system to change its state or to perform a specific action. Examples of events are: arrival of a packet, a periodic alarm informing a routing protocol to send out routing update to neighbors, etc. Examples of actions to take when an event occurs are: sending a packet to an adjacent layer, updating state variables, starting or restarting a timer, etc.

In discrete-event simulation, the simulator maintains an *event queue*. Associated with each event is its *event time*, i.e., the time at which the event is set to occur. Events in the event queue are sorted by the event time. The simulator also maintains a *simulation clock* which is used to simulate time. The simulation clock is advanced in discrete steps, as explained below.

The simulator operates by continually repeating the following series of steps until the end of simulation:

- The simulator removes the first event from the event queue, i.e., the event scheduled for the earliest time.
- The simulator sets the simulation clock to the event time of the event. This may result in advancing the simulation clock.
- The simulator handles the event, i.e., it executes the actions associated with the event. This may result in changing the system state, scheduling other events, or both. If other events are scheduled, they may be scheduled to occur at the current time or in the future.

3.2 Modeling Protocols in EXata

As discussed in Section 1.2, each node in EXata runs a protocol stack, shown in Figure 1-1. Each layer provides a service to the layer above it, by using the services of the layers below it.

Each protocol operates at one of the layers of the stack. Protocols in EXata essentially operate as a finite state machine. The occurrence of an event corresponds to a transition in the finite state machine. The interface between the layers is also event based. Each protocol can either create events that make it change its own state (or perform some event handling), or create events that are processed by another protocol. To pass data to, or request a service from, an adjacent layer, a protocol creates an event for that layer.

Figure 3-1 shows the finite state machine representation of a protocol in EXata. At the heart of a protocol model is an *Event Dispatcher*, which consists of a *Wait For Event* state and one or more *Event Handler* states (see Figure 3-1). In the *Wait For Event* state, the protocol waits for an event to occur. When an event for the protocol occurs, the protocol transitions to the *Event Handler* state corresponding to that event (e.g., when *Event 1* occurs, the protocol transitions to the *Event 1 Handler* state). In this *Event Handler* state, the protocol performs the actions corresponding to the event, and then returns to the *Wait For Event* state. Actions performed in the *Event Handler* state may include updating the protocol state, or scheduling other events, or both.

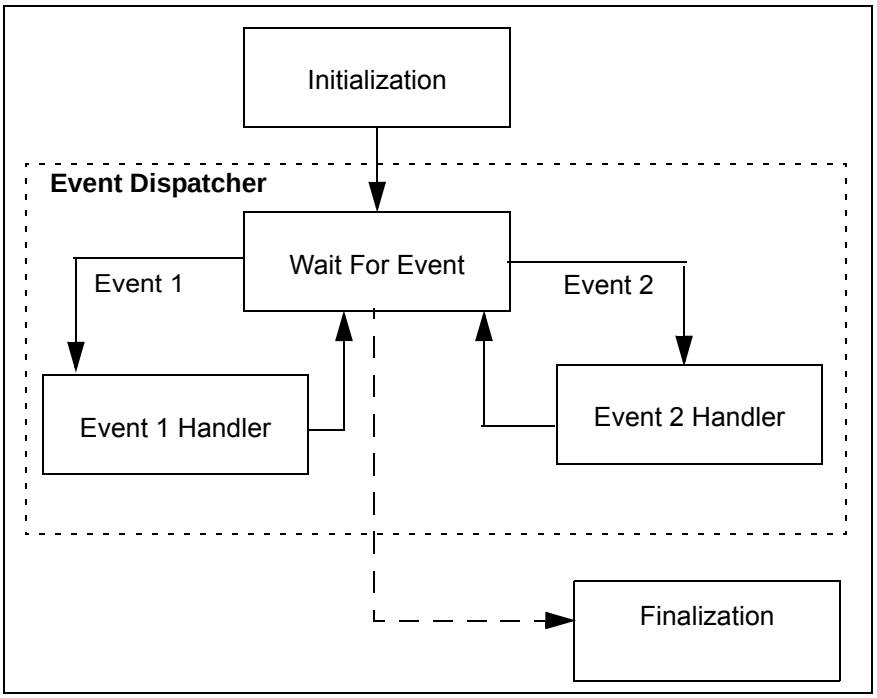


FIGURE 3-1. Protocol Model in EXata

Besides the *Event Dispatcher*, the protocol finite state machine has two other states: the *Initialization* state and the *Finalization* state. In the *Initialization* state, the protocol reads external input to configure its initial state. The protocol then transitions to the *Wait For Event* state.

The transition to the *Finalization* state occurs automatically at the end of simulation. In the *Finalization* state, protocol statistics collected during the simulation are printed.

3.3 Discrete-event Simulation in EXata

This section describes implementation details of discrete-event simulation in EXata: types of events, data structures and classes to implement events, and API functions for event operations.

3.3.1 Events and Messages

In EXata, the class used to represent an event is called a *message*. A message holds information about the event such as the type of event, and the associated data. In the context of EXata, the terms *event* and *message* are often used interchangeably.

There are two types of events in EXata: *packet events* and *timer events*. Packet events are used to simulate exchange of data packets between layers or between nodes. Packet events are also used for modeling communication between different entities at the same layer. Timer events are used to simulate time-outs and are internal to a protocol. Packets events are discussed in Section 3.3.2.1 and timer events are discussed in Section 3.3.2.2.

In this section, we describe the message class and the APIs for message operations.

3.3.1.1 Message Class

The class `Message`, defined in file EXATA_HOME/include/message.h, is used to implement events. Figure 3-2 shows the main components of this class. Both packet and timer events are implemented using the `Message` class.

```
class Message
{
private:
    static const UInt8 SENT = 0x01; // Message is being sent
    ...
public:
    // The default constructor should not be used unless under specific
    // circumstances.  The message is not initialized here.
    Message();
  ...
  short layerType;          // Layer which will receive the message.
  short protocolType;       // Protocol which will receive the
                            // message in the layer.
  short instanceId;         // Which instance to give message to (for multiple
                            // copies of a protocol or application).
  short eventType;          // Message's event type.
  ...
  int packetSize;           // Size of the packet field.
  char *packet;             // Simulates a data packet, including headers.
  ...
  int virtualPayLoadSize;  // Size of "virtual" data.

  clocktype packetCreationTime;   // If this is a packet, it's creation time.
  ...
  std::vector<MessageInfoHeader> infoArray;
  ...
}
```

FIGURE 3-2. Message Class

Some of the members of the `Message` class are explained below.

- `layerType`: This is the layer associated with the event.
- `protocolType`: This is the protocol associated with the event.
- `instanceId`: If there are multiple instances of a protocol, this field denotes the instance of the protocol associated with the event.
- `eventType`: This is the type of the event. Event types are listed in EXATA_HOME/include/api.h.
- `packet`: If the class instance is used to simulate an actual data packet in the network, this field stores the packet. Headers added by different layers are included in this field.
- `packetSize`: This is the size of the `packet` field.
- `virtualPayLoadSize`: This is the size of that part of user data whose contents are not important and hence is not allocated any memory, but whose size affects the calculation of transmission time and buffer size.
- `packetCreationTime`: If the class instance is used to simulate an actual packet in the network, this field stores the packet's creation time.
- `infoArray`: This is an array that stores additional information that is used in the processing of the event and for information that needs to be transported between layers or nodes. See Section 3.3.1.1.1 for details of this field.

3.3.1.1.1 Message `infoArray` Member

The message `infoArray` member is an array used to store extra information about the message that is used for processing the message as well as information that needs to be transported between layers or nodes. This information does not affect the transmission delay calculations because it does not model the actual data being transmitted.

Each element of the array is a structure of type `MessageInfoHeader`, which is declared in EXATA_HOME/include/message.h and is shown in Figure 3-3.

```
typedef struct message_info_header_str
{
    unsigned short infoType; // type of the info field
    unsigned short infoSize; // size of buffer pointed to by "info" variable
    char* info;              // pointer to buffer for holding info
} MessageInfoHeader;
```

FIGURE 3-3. `MessageInfoHeader` Data Structure

The fields of the MessageInfoHeader data structure are explained below.

- infoType: This indicates the type of the information contained in this struct. The value of this field can be one of the enumerations of the type MessageInfoType which is declared in EXATA_HOME/include/message.h and is shown in Figure 3-4. Users can add additional members to this enumeration for their use, as explained in Section 4.1.2.
- infoSize: This is the size of the info field of this struct.
- info: This is the pointer to the buffer that stores the information.

```
typedef enum message_info_type_str
{
    INFO_TYPE_UNDEFINED = 0,  // an empty info field.
    INFO_TYPE_DEFAULT = 1,    // default info type used in situations where
                              // specific type is given to the info field.
    INFO_TYPE_AbstractCFPropagation, // type for abstract contention free
                                     // propagation info field.
    INFO_TYPE_AppName,      // Pass the App name down to IP layer
    INFO_TYPE_StatCategoryName,
    INFO_TYPE_DscpName,
    ...
    INFO_TYPE_TransStatsDbContent,
    INFO_TYPE_NetStatsDbContent,
    ...
} MessageInfoType;
```

FIGURE 3-4. MessageInfoType Enumeration Type

Different elements of the infoArray field can be used for different purposes, for both timer events and packet events. For example, one of the elements of the array can be used to store additional information associated with a timer, e.g., for a timer indicating that a route has expired, the destination address for the expired route can be stored in one of the elements of infoArray. This can assist the timer event handler to locate and remove the correct entry in the routing table.

EXata provides several APIs to manipulate the infoArray field. See Section 4.1.2 for a more detailed description of the infoArray field and its associated APIs.

3.3.1.1.2 Message packet Field

The message packet field simulates the actual data being transmitted. Unlike the infoArray field, the size of this field, indicated by the packetSize member of the Message class, does affect the transmission delay calculations.

3.3.1.2 Message APIs

Several API functions are available in EXata for message operations. The message APIs can be called from any layer. The prototypes for these functions can be found in the file EXATA_HOME/include/message.h. The implementation code for these functions can be found in the file EXATA_HOME/main/message.cpp. Some of the message APIs are listed below. Refer to *API Reference Guide* or the file message.h for a complete list of message APIs and their parameters.

- MESSAGE_Send: This function schedules the specified event (message) to occur after the specified delay.

Do not alter any fields of the message class instance after an event has been scheduled by calling MESSAGE_Send.

- MESSAGE_Alloc: This function allocates a new message structure and sets the `layerType`, `protocolType` and `eventType` fields of the structure to the values passed to the function as parameters.
- MESSAGE_Free: This function frees the message specified. The `packet` and `infoArray` fields of the message are freed, and then the message itself is freed.
- MESSAGE_AddInfo: This function allocates one element of the `infoArray` field of the specified message. The type and size of the associated `info` field are passed as parameters.
- MESSAGE_ReturnInfo: This function takes the `infoType` as a parameter and returns a pointer to the associated `info` field of the specified message.
- MESSAGE_ReturnInfoSize: This function takes the `infoType` as a parameter and returns the associated `infoSize` field of the specified message.
- MESSAGE_PacketAlloc: This function allocates the `packet` field of the specified message. The size of the `packet` field and the name of the protocol that creates the packet are passed as parameters.
- MESSAGE_ReturnPacket: This function returns a pointer to the `packet` field of the specified message.
- MESSAGE_ReturnActualPacketSize: This function returns the `packetSize` field of the specified message.
- MESSAGE_CancelSelfMsg: This function cancels a message that had been scheduled earlier.

Do not free the message explicitly or re-use the message after canceling it. The function MESSAGE_CancelSelfMsg also frees the memory associated with the message.

- MESSAGE_AddHeader: This function adds a header to the packet enclosed in the specified message. The `packetSize` field of the message is incremented by the size of the header and the `packet` field points to the newly allocated header. The header size and the name of the protocol that adds the header are passed as a parameters.
- MESSAGE_RemoveHeader: This function removes a header from the packet enclosed in the specified message. The `packetSize` field of the message is decremented by the size of the header, and the `packet` field points to the space after the removed header. The header size and the name of the protocol that removes the header are passed as parameters.
- MESSAGE_GetLayer: This function returns the `layerType` field of the specified message.
- MESSAGE_GetProtocol: This function returns the `protocolType` field of the specified message.
- MESSAGE_GetEvent: This function returns the `eventType` field of the specified message.

3.3.2 Types of Events

In EXata, there are two types of events: packet events and timer events. Although both packet and timer events are defined using the same `Message` class (see Section 3.3.1.1), they vary in their purpose and the manner in which they are handled by EXata.

3.3.2.1 Packet Events

Packet events are used to simulate transmission of packets across the network. A packet is defined as a unit of virtual or real data at any layer of the protocol stack. When a node needs to send a packet to an adjacent layer in the EXata protocol stack, it schedules a packet event at the adjacent layer. The occurrence of the packet event at the adjacent layer simulates the arrival of the packet.

When a protocol residing at a particular layer at one node sends packets the corresponding protocol at the same layer at another node, the packet is passed down through the protocol stack at the sending node, across the network, and then up through the protocol stack at the receiving node. At each level of the protocol stack at the sending node, header information is added to the packet as it is sent to the layer below. Each layer is responsible for sending the packet to its adjacent layer. At the receiving node, each layer strips off its header and sends the packet to the layer above, until the original packet is finally available to the receiving protocol. Figure 3-5 shows an example of this process for the case when the originating protocol resides at the Application Layer. The steps in this process are listed below.

- The originating protocol creates a new message by using the API MESSAGE_Alloc. The protocol creates the `packet` field of this message by using the API MESSAGE_PacketAlloc.
- The protocol puts the data to be sent to the receiving node in the `packet` field of the message, sets the other fields of the message appropriately, and sends the message to the next layer (Transport Layer in this case) by using the API MESSAGE_Send. Function MESSAGE_Send schedules a packet event for the next layer to occur after a delay that is specified as a parameter.
- When the packet is received by the Transport Layer protocol, the Transport Layer protocol appends its header to the packet by using the API MESSAGE_AddHeader and sets the header fields appropriately. The Transport Layer protocol then sends the resulting packet to the next layer in the stack by using the API MESSAGE_Send.
- The previous step is repeated at each layer in the protocol stack: Each layer adds its header to the packet and sends the resulting packet to the next layer.
- When the packet arrives at the Physical Layer of the source node, it schedules a packet receive event for the Physical Layer at the destination node.
- When a layer at the destination node receives a packet, it removes the corresponding header using the API MESSAGE_RemoveHeader, and sends the resulting packet to the next higher layer in the protocol stack using the API MESSAGE_Send.
- The previous step is repeated at each layer in the protocol stack: Each layer removes its header and sends the resulting packet to the next higher layer.
- When the packet arrives at the Application Layer at the destination node, the receiving protocol processes the packet and frees the message using the API MESSAGE_Free.

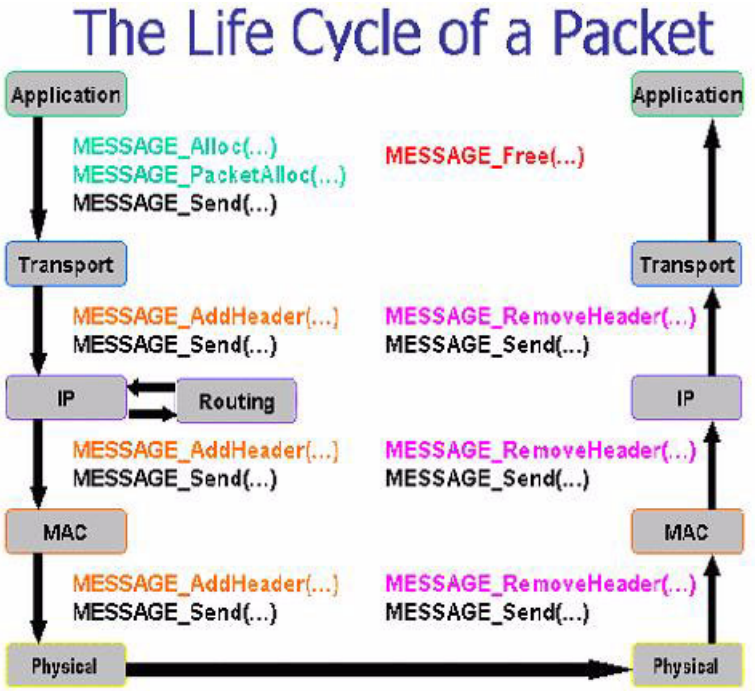


FIGURE 3-5. The Life Cycle of a Packet

In EXata, communication between adjacent layers can take place by using message APIs, as shown in Figure 3-5, or by using layer-specific APIs. Message APIs are generic and can be used at any layer, while layer-specific APIs are specific to a particular layer. Section 3.3.2.1.1 gives an overview of the use of layer-specific APIs for packet exchange between layers. Section 3.3.2.1.2 gives an overview of the use of message APIs for packet exchange between layers

3.3.2.1.1 Sending Packets Using Layer-specific APIs

To simplify protocol development, EXata provides layer-specific API functions to send packets. Rather than using the raw message API as shown in Figure 3-5, the protocol developer can simply use the layer-specific API functions to send packets from a particular layer. The layer-specific API functions are responsible for scheduling events at the adjacent layers as a packet travels through the protocol stack. The APIs provided at each layer encapsulate the message APIs and hides details of scheduling events at the adjacent layer, thus providing easy-to-use functions for sending out packets from a specific layer of the protocol stack.

The layer-specific API functions vary for each layer. The API calls available at each layer are discussed in the corresponding section of Chapter 4. To understand the available APIs at each layer, refer to the API functions used for sending packets in the source code of EXata implementation of protocols operating at that layer. As an example, we give an overview of using the layer-specific packet exchange APIs available at the Application Layer in this section.

Packet exchanges at the Application Layer fall into two categories:

- Exchanging packets with the UDP protocol at the Transport Layer
- Exchanging packets with the TCP protocol at the Transport Layer

Table 3-1 lists the API calls available for sending packets at the Application Layer using UDP at the Transport Layer. Table 3-2 lists the API calls available for sending packets at the Application Layer using TCP at the Transport Layer. These functions are defined in EXATA_HOME/main/app_util.cpp. The underlying code for each of these functions creates and sends messages using the message APIs discussed in Section 3.3.1.2.

TABLE 3-1. API Functions for Sending Packets via UDP

API Function	Description
APP_UdpSendNewData	Sends user data via UDP to a destination after a user-specified delay.
APP_UdpSendNewDataWithPriority	Sends user data via UDP to a destination after a user-specified delay with a user-specified priority value.
APP_UdpSendNewHeaderData	Appends an application header to user data and sends it via UDP to a destination after a user-specified delay.
APP_UdpSendNewHeaderDataWithPriority	Appends an application header to user data and sends it via UDP to a destination after a user-specified delay with a user-specified priority
APP_UdpSendNewHeaderVirtualDataWithPriority	Sends a packet composed of an application header containing useful information and user data whose contents are unimportant and serve only to add to resource consumption (queue capacity, transmission delays, etc.), via UDP to a destination after a user-specified delay with a user-specified priority value.

Note **The APP_UdpSendNewHeaderVirtualDataWithPriority function is overloaded and can also be used to send to a particular port number.**

TABLE 3-2. API Functions for Sending Packets via TCP

API Function	Description
APP_TcpSendData	Sends user data via TCP to a destination.
APP_TcpSendNewHeaderVirtualData	Sends a packet composed of an application header containing useful information and user data whose contents are unimportant and serve only to add to resource consumption (queue capacity, transmission delays, etc.), via TCP to a destination.

Figure 3-6 shows a code segment from the RIP implementation function RipSendResponse that uses the API function APP_UdpSendNewDataWithPriority to send a packet from the Application Layer. Function RipSendResponse is implemented in EXATA_HOME/libraries/developer/src/routing_rip.cpp. Notice that initially a variable, `response,` is defined. This variable is the user data. This variable is then filled with information (RIP command and RIP version information). Next, the function assigns the destination address and calls the API function APP_UdpSendNewDataWithPriority to send the packet. The parameters of this API function are explained below.

- `node`: Pointer to the node
- `appType`: Application type
- `sourceAddr`: Source address
- `sourcePort`: Source port number
- `destAddr`: Destination address
- `outgoingInterface`: Outgoing interface index
- `payload`: Pointer to user data
- `payloadSize`: Size of user data
- `TosType`: Priority for the packet
- `delay`: Delay after which data is to be sent
- `traceProtocol`: Trace protocol

```
static void RipSendResponse(Node* node, int interfaceIndex,
                            RipResponseType type)
{
    ...
    unsigned routeIndex;
    RipResponse response;
    response.command = RIP_RESPONSE;       // Response message
    ...
    routeIndex = 0;
    while (routeIndex < dataPtr->numRoutes)
    {
        int rteIndex;
        ...
        if (rteIndex != 0)
        {
            NodeAddress destAddress;
            if (NetworkIpIsWiredNetwork(node, interfaceIndex))
            {
                destAddress = NetworkIpGetInterfaceBroadcastAddress(
                                    node,
                                    interfaceIndex);
            }
            else
            {
                destAddress = ANY_DEST;
            }

            APP_UdpSendNewDataWithPriority(
                node,
                APP_ROUTING_RIP,
                NetworkIpGetInterfaceAddress(node, interfaceIndex),
                APP_ROUTING_RIP,
                destAddress,
                interfaceIndex,
                (char*) &response,
                RIP_HEADER_SIZE + RIP_RTE_SIZE * rteIndex,
                IPTOS_PREC_INTERNETCONTROL,
                RANDOM_nrand(dataPtr->updateSeed)
                                        % (clocktype) RIP_STARTUP_DELAY,
                TRACE_RIP);
            ...
            }
        }
}
```

FIGURE 3-6. Sending a Packet Using a Layer-specific API

3.3.2.1.2 Sending Packets Using Message APIs

Layer-specific APIs provide a convenient means for sending packets through the protocol stack. However, sometimes it may be desirable to bypass the layer-specific APIs. This may be due to certain specifics of protocol design. This section describes how to send packets using message APIs.

To understand the use of message APIs, take a look at the implementation of the layer-specific API, APP_UdpSendNewDataWithPriority, used in function RipSendResponse (see Section 3.3.2.1.1) to send a packet from the Application Layer to the UDP protocol at the Transport Layer. APP_UdpSendNewdataWithPriority is implemented in EXATA_HOME/main/app_util.cpp and is shown in Figure 3-7.

```
void
APP_UdpSendNewDataWithPriority(
    Node *node,
    AppType appType,
    NodeAddress sourceAddr,
    short sourcePort,
    NodeAddress destAddr,
    int outgoingInterface,
    char *payload,
    int payloadSize,
    TosType priority,
    clocktype delay,
    TraceProtocolType traceProtocol)
{
    Message *msg;
    AppToUdpSend *info;
    ActionData acnData;

    msg = MESSAGE_Alloc(
              node,
              TRANSPORT_LAYER,
              TransportProtocol_UDP,
              MSG_TRANSPORT_FromAppSend);

    MESSAGE_PacketAlloc(node, msg, payloadSize, traceProtocol);

    memcpy(MESSAGE_ReturnPacket(msg), payload, payloadSize);
    MESSAGE_InfoAlloc(node, msg, sizeof(AppToUdpSend));
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);
    SetIPv4AddressInfo(&info->sourceAddr, sourceAddr);
    info->sourcePort = sourcePort;
    SetIPv4AddressInfo(&info->destAddr, destAddr);
    info->destPort = (short) appType;
    info->priority = priority;
    info->outgoingInterface = outgoingInterface;
    info->ttl = IPDEFTTL;

    //Trace Information
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node, msg, TRACE_APPLICATION_LAYER,
        PACKET_OUT, &acnData);

    MESSAGE_Send(node, msg, delay);
}
```

FIGURE 3-7. Implementation of APP_UdpSendNewDataWithPriority

Function APP_UdpSendNewDataWithPriority allocates a `message` variable `msg` using the API MESSAGE_Alloc. This is followed by a call to MESSAGE_PacketAlloc to allocate the `packet` field of the message. The third parameter to MESSAGE_PacketAlloc, `payloadSize`, is used to set the size of the `packet` field. Once MESSAGE_PacketAlloc has been called, the `packet` field in the message structure can be used to access this space. The API function MESSAGE_ReturnPacket is used to access the `packet` field of the message.

The user data is then copied into the `packet` field using the memcpy function. Additional information can be stored in the `infoArray[0].info` field of the message (see Section 3.3.1.1.1), which is allocated using the API MESSAGE_InfoAlloc. (MESSGE_InfoAlloc is equivalent to using MESSAGE_AddInfo with `INFO_TYPE_DEFAULT` as the `info` field type, and allocates the 0^{th} element of `infoArray`). The API MESSAGE_ReturnInfo is used to access the `infoArray[0].info` field of the message. After storing information in the `infoArray[0].info` field of the message, the packet is sent to the next layer using the MESSAGE_Send function. (When the message is allocated using MESSAGE_Alloc in the first step of APP_UdpSendNewDataWithPriority, the `layerType`, `protocolType` and `eventType` fields are set to `TRANSPORT_LAYER`, `TransportProtocol_Udp` and `MSG_TRANSPORT_FromAppSend`, respectively. The result of calling MESSAGE_Send in the last step of APP_UdpSendNewDataWithPriority is to schedule a `MSG_TRANSPORT_FromAppSend` event at the UDP protocol at the Transport Layer after a delay specified by the third parameter of MESSAGE_Send.)

When a packet from the Application Layer arrives at the UDP protocol at the Transport Layer, UDP appends a header to the packet and sends it to the next layer (Network Layer). This is done in the UDP function TransportUdpSendToNetwork, which is implemented in EXATA_HOME//libraries/developer/src/transport_udp.cpp and is shown in Figure 3-8.

```
void
TransportUdpSendToNetwork(Node *node, Message *msg)
{
    TransportDataUdp *udp = (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader *udpHdr;
    AppToUdpSend *info;

    if (udp->udpStatsEnabled == TRUE)
    {
        udp->statistics->numPktFromApp++;
    }

    MESSAGE_AddHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);

    udpHdr = (TransportUdpHeader *) msg->packet;
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);

    udpHdr->sourcePort = info->sourcePort;
    udpHdr->destPort = info->destPort;
    udpHdr->length = (unsigned short) MESSAGE_ReturnPacketSize(msg);
    udpHdr->checksum = 0;  /* checksum not calculated */

    ActionData acnData;
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_OUT,
                     &acnData);

    NetworkIpReceivePacketFromTransportLayer(
        node,
        msg,
        info->sourceAddr,
        info->destAddr,
        info->outgoingInterface,
        info->priority,
        IPPROTO_UDP,
        FALSE,
        info->ttl);
}
```

FIGURE 3-8. Adding a Header at Transport Layer

In function TransportUdpSendToNetwork, the API function MESSAGE_AddHeader is used to add a header before a packet. This function reserves additional space in the packet for a header. The header size is specified by the third parameter of the function. MESSAGE_AddHeader also appropriately updates the `packetSize` field in the `message` structure. After this function is called, the `packet` field in the `message` structure points to the space occupied by this new header.

TransportUdpSendToNetwork next updates the header fields and calls the Transport Layer-specific API function NetworkIpReceivePacketFromTransportLayer to send the packet to the next layer (the Network Layer). In this way the packet travels down the protocol stack with each layer adding its own header. This is graphically illustrated in Figure 3-5.

3.3.2.2 Timer Events

Timer events are used to perform the function of alarms. They essentially allow an application to schedule events for itself at a future time. Periodic alarms are implemented by re-setting the timer event after it has occurred. Timer events are set and received within a protocol and they do not travel through the protocol stack.

Examples of timer events are:

- Timer alarm to send route updates every 5 seconds
- Timer alarm to remove expired route from routing table 3 seconds after it is installed

3.3.2.2.1 Setting Timers

Timer events are also implemented using the `message` class described in Section 3.3.1.1. To set a timer event, allocate a new message using the function MESSAGE_Alloc (see Section 3.3.1.2). Pass as parameters to the function the node pointer, the layer, the protocol and the event type. The event types are defined in EXATA_HOME/include/api.h.

For example, the following code schedules an event of type `MSG_APP_RIP_RegularUpdateAlarm` for the RIP protocol at the Application Layer to occur after a delay of 5 seconds from the current simulation time.

```
Message *newMsg;
clocktype delay;
newMsg = MESSAGE_Alloc(node,
                       APP_LAYER,
                       APP_ROUTING_RIP,
                       MSG_APP_RIP_RegularUpdateAlarm);
delay = 5* SECOND;
MESSAGE_Send(node, newMsg, delay);
```

Note that if the delay is set to 0, the event occurs after the current function finishes execution but before the simulation clock is advanced.

It may be required to store some additional information with a timer. The message `infoArray` field is used with timers for this purpose. As an example, consider a time-out timer to receive an acknowledgment for a transmitted packet. In this case, the `infoArray[0].info` field of the message can store the sequence number and destination IP address of the packet for which an acknowledgment is expected. See Section 3.3.1.1.1 for more details of the message `infoArray` field.

3.3.2.2.2 Canceling Timers

The API function MESSAGE_CancelSelfMsg (see Section 3.3.1.2) is used to cancel a message in the EXata scheduler. The message must be a self message, i.e., a message the node sent to itself. The function accepts a pointer to a node and a pointer to the message to be cancelled as arguments.

For example, consider the following function call:

```
MESSAGE_CancelSelfMsg(node, msgToCancelPtr)
```

In the function call above, *msgToCancelPtr* is a pointer to the original message that needs to be canceled. To use this function, the pointer to the original message has to be retained.

3.4 EXata Simulator Architecture

As discussed in Section 3.2, a protocol model in EXata has three components: Initialization, Event Handling, and Finalization. Each of these functions is performed hierarchically: first at the node level, then at the layer level, and finally at the protocol level. The following sections describe the hierarchy of these three functions.

3.4.1 Initialization Hierarchy

At the start of simulation, each node in the network is initialized. Function PARTITION_InitializeNodes, defined in EXATA_HOME/main/partition.cpp and shown in Figure 3-9, is the function which initializes nodes. Function PARTITION_InitializeNodes initializes the layers of the protocol stack running at every node by calling the initialization function for each layer. The layers are initialized in a bottom-up order, starting from the bottom-most layer. Some layers, such as the MAC Layer, are initialized globally, while the other layers are initialized one node at a time. For example, function MAC_Initialize initializes the MAC Layer for all nodes, while function TRANSPORT_Initialize initializes the Transport Layer at a given node. There are two initialization functions for the Application Layer: one for traffic-generating protocols and the other for routing protocols running at the Application Layer (these are discussed in detail in Section 4.2). Function APP_Initialize initializes the Application Layer routing protocols for a given node, and function APP_InitializeApplications initializes the Application Layer traffic-generating protocols at all nodes.

```
void PARTITION_InitializeNodes(PartitionData* partitionData)
{
    int             i, j;
    Node*           nextNode  = NULL;
    ...
    // Initalize global antenna model
    ANTENNA_GlobalAntennaModelPreInitialize(partitionData);
    ANTENNA_GlobalAntennaPatternPreInitialize(partitionData);
    ...
    nextNode = partitionData->firstNode;
    while (nextNode != NULL) {
        ...
        NETWORK_PreInit(nextNode, nodeInput);
        PHY_Init(nextNode, nodeInput);
        ...
        nextNode = nextNode->nextNodeData;
    }
    ...
    // Initialize globally, rather than a node at a time.
    MAC_Initialize(partitionData->firstNode, nodeInput);
    ...
    nextNode = partitionData->firstNode;
    BOOL wasFound;
    char name[MAX_STRING_LENGTH];
    while (nextNode != NULL)
    {
        ...
        NETWORK_Initialize(nextNode, nodeInput);
        TRANSPORT_Initialize(nextNode, nodeInput);
        APP_Initialize(nextNode, nodeInput);
        ...
        nextNode = nextNode->nextNodeData;
    }

    // Initialize globally, rather than a node at a time.
    APP_InitializeApplications(partitionData->firstNode,
                               nodeInput);
    ...
}
```

FIGURE 3-9. Node Initialization Function

Each layer initialization function, in turn, calls an initialization function for each protocol running at that layer. For example, function TRANSPORT_Initialize, defined in EXATA_HOME/main/transport.cpp and shown in Figure 3-10, calls the initialization functions for the TCP and UDP protocols, TransportTcpInit and TransportUdpInit, respectively. Function TransportTcpInit is defined in EXATA_HOME/libraries/developer/src/transport_tcp.cpp and function TransportUdpInit is defined in EXATA_HOME/libraries/developer/src/transport_udp.cpp.

```
void TRANSPORT_Initialize(Node * node,
                          const NodeInput * nodeInput)
{
    ...

    node->transportData.tcp = NULL;
    node->transportData.udp = NULL;

    TransportTcpInit(node, nodeInput);
    TransportUdpInit(node, nodeInput);
    ...
 }
```

FIGURE 3-10. Layer Initialization Function

The initialization function of a protocol creates and initializes the protocol state variables, as well as the protocol statistics variables. For example, Figure 3-11 shows the initialization function for the UDP protocol, TransportUdpInit. Function TransportUdpInit creates the UDP state variable `udp`, which is a data structure of type `TransportDataUdp`. If UDP statistics collection is enabled, TransportUdpInit also creates and initializes the UDP statistics variable, which is a data structure of type `TransportUdpStat`. `TransportDataUdp` and `TransportUdpStat` are defined in EXATA_HOME/include/transport.h and EXATA_HOMElibraries/developer/src/transport_udp.h, respectively.

```
void TransportUdpInit(Node *node, const NodeInput *nodeInput)
{
    char buf[MAX_STRING_LENGTH];
    BOOL retVal;
    TransportDataUdp* udp =
                   (TransportDataUdp*)
                    MEM_malloc(sizeof(TransportDataUdp));
    node->transportData.udp = udp;
    TransportUdpInitTrace(node, nodeInput);
    IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "UDP-STATISTICS",
        &retVal,
        buf);

  if (retVal == FALSE || strcmp(buf, "NO") == 0)
    {
        udp->udpStatsEnabled = FALSE;
    }
    else if (strcmp(buf, "YES") == 0)
    {
        udp->udpStatsEnabled = TRUE;
    }
    else
    {
        ...
    }
    if (udp->udpStatsEnabled == TRUE)
    {
        udp->statistics = (TransportUdpStat *)
                          MEM_malloc(sizeof(TransportUdpStat));
        ...
        memset(udp->statistics, 0, sizeof(TransportUdpStat));
        ...;
    }
}
```

FIGURE 3-11. Protocol Initialization Function

3.4.2 Event Handling Hierarchy

When an event occurs, the EXata kernel gets a handle to the node for which the event is scheduled. It then calls a dispatcher function, NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp and shown in Figure 3-12. This function determines the layer for which the event has occurred and calls the event dispatcher function for the appropriate layer, e.g., if the event is for the Application Layer, NODE_ProcessEvent calls the Application Layer event dispatcher function, APP_ProcessEvent.

```
void NODE_ProcessEvent(Node *node, Message *msg)
{
    ...
    switch (MESSAGE_GetLayer(msg))
    {
        case PROP_LAYER:
        {
            ...
            PROP_ProcessEvent(node, msg);
            ...
            break;
        }
        case PHY_LAYER:
        {
            PHY_ProcessEvent(node, msg);
            break;
        }
        case MAC_LAYER:
        {
            MAC_ProcessEvent(node, msg);
            break;
        }
        case NETWORK_LAYER:
        {
            NETWORK_ProcessEvent(node, msg);
            break;
        }
        case TRANSPORT_LAYER:
        {
            TRANSPORT_ProcessEvent(node, msg);
            break;
        }
        case APP_LAYER:
        {
            APP_ProcessEvent(node, msg);
            break;
        }
        ...
    }
}
```

FIGURE 3-12. Node Event Handler Function

The event dispatcher function for a layer determines the protocol for which the event has occurred, and calls the event handler for that protocol. For example, when an event for the Bellman-Ford protocol occurs, the Application Layer dispatcher function, APP_ProcessEvent, calls function RoutingBellmanfordLayer, which is the event handler for the Bellman-Ford protocol. This is illustrated in Figure 3-13. Function APP_ProcessEvent is defined in EXATA_HOME/main/application.cpp.

```
void APP_ProcessEvent(Node *node, Message *msg)
{
    short protocolType;
    protocolType = APP_GetProtocolType(node,msg);

    switch(protocolType)
    {
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
        case APP_ROUTING_FISHEYE:
        {
            RoutingFisheyeLayer(node,msg);
            break;
        }
        ..
        case APP_FTP_CLIENT:
        {
            AppLayerFtpClient(node, msg);
            break;
        }
        case APP_FTP_SERVER:
        {
            AppLayerFtpServer(node, msg);
            break;
        }
       ...
    }//switch//
}
```

FIGURE 3-13. Layer Event Dispatcher Function

The protocol event dispatcher, like the other dispatcher functions, consists of a switch statement. It calls the event handler function for the event that has occurred. An event handler is specific to an event and performs the required actions on the occurrence of that event. For example, the Bellman-Ford dispatcher function, RoutingBellmanfordLayer, shown in Figure 3-14, calls function HandleFromTransport when an event of type `MSG_APP_FromTransport` occurs. `MSG_APP_FromTransport` indicates that a packet has been received from the Transport Layer, and function HandleFromTransport performs the actions required to handle the received packet. Functions RoutingBellmanfordLayer and HandleFromTransport are defined in EXATA_HOME/libraries/developer/src/routing_bellmanford.cpp.

```
void RoutingBellmanfordLayer(Node *node, Message *msg)
{
    if (node->networkData.networkProtocol == IPV6_ONLY)
    {
        // Bellmanford is an IPv4 Network based routing protocol,
        // it can not be run on this node
        ...
        MESSAGE_Free(node, msg);

        return;
    }
    switch(msg->eventType)
    {
        case MSG_APP_PeriodicUpdateAlarm:
        {
            HandlePeriodicUpdateAlarm(node);
            break;
        }
        case MSG_APP_CheckRouteTimeoutAlarm:
        {
            HandleCheckRouteTimeoutAlarm(node);
            break;
        }
        case MSG_APP_TriggeredUpdateAlarm:
        {
            HandleTriggeredUpdateAlarm(node);
            break;
        }

        // Messages sent by UDP to Bellman-Ford.

        case MSG_APP_FromTransport:
        {
            HandleFromTransport(node, msg);
            break;
        }
        default:
            ERROR_ReportError("Invalid switch value");
    }

    // Done with the message, so free it.

    MESSAGE_Free(node, msg);
}
```

FIGURE 3-14. Protocol Event Dispatcher Function

3.4.3 Finalization Hierarchy

At the end of simulation, the finalization function for each protocol is called to print the protocol statistics. Like the initialization and event handling functions, the finalization function is called hierarchically.

Figure 3-15 shows the node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp. PARTITION_Finalize calls the finalization function for each layer in the protocol stack running at each node. For example, MAC_Finalize is the finalization function for the MAC Layer.

```
void PARTITION_Finalize(PartitionData* partitionData)
{
    if (partitionData->firstNode != NULL)
    {
        Node *nextNode = partitionData->firstNode;
        while (nextNode != NULL)
        {
            PHY_Finalize(nextNode);
            MAC_Finalize(nextNode);
            ...
            if ((nextNode->adaptationData.adaptationProtocol
                == ADAPTATION_PROTOCOL_NONE)
                || (nextNode->adaptationData.endSystem))
            {
                NETWORK_Finalize(nextNode);
                TRANSPORT_Finalize(nextNode);
                APP_Finalize(nextNode);
                USER_Finalize(nextNode);
                MOBILITY_Finalize(nextNode);
            }
            nextNode = nextNode->nextNodeData;
        }
    }
    ...
}
```

FIGURE 3-15. Node Finalization Function

The finalization function for a layer calls the finalization function for each protocol running at that layer. For example, consider the MAC Layer finalization function, MAC_Finalize, defined in EXATA_HOME/main/mac.cpp and shown in Figure 3-16. For each interface of a node, MAC_Finalize calls the finalization function for the MAC protocol running at that interface, e.g., if the CSMA protocol is running at an interface, MAC_Finalize calls the CSMA finalization function MacCsmaFinalize.

```
void MAC_Finalize(Node *node)
{
    int interfaceIndex;
    ...
    for (interfaceIndex = 0;
         interfaceIndex < node->numberInterfaces;
         interfaceIndex++)
    {
        /* Select the MAC protocol model and finalize it. */

        if (node->macData[interfaceIndex])
        {
            switch
                (node->macData[interfaceIndex]->macProtocol)
            {
                case MAC_PROTOCOL_DOT11:
                {
                    MacDot11Finalize(node, interfaceIndex);
                    break;
                }
                case MAC_PROTOCOL_CSMA:
                {
                    MacCsmaFinalize(node, interfaceIndex);
                    break;
                }
                ...
             }
             ...
       }
       ...
}
```

FIGURE 3-16. Layer Finalization Function

The finalization function for a protocol prints the statistics for the protocol if statistics collection is enabled for the layer in which the protocol resides. For example, function MacCsmaFinalize, shown in Figure 3-17, calls the function to print CSMA statistics, MacCsmaPrintStats, if statistics collection is enabled for the MAC Layer. Functions MacCsmaFinalize and MacCsmaPrintStats are defined in EXATA_HOME/libraries/wireless/src/mac_csma.cpp.

```
void MacCsmaFinalize(Node *node, int interfaceIndex)
{
    MacDataCsma* csma = (MacDataCsma *)
                        node->macData[interfaceIndex]->macVar;

    if (node->macData[interfaceIndex]->macStats == TRUE) {
        MacCsmaPrintStats(node, csma, interfaceIndex);
    }
}
```

FIGURE 3-17. Protocol Finalization Function

4 Developing Protocol Models in EXata

The EXata protocol stack, shown in Figure 4-1, is similar to the TCP/IP protocol stack and consists of the following five layers:

- Application Layer
- Transport Layer
- Network Layer
- MAC Layer
- Physical Layer

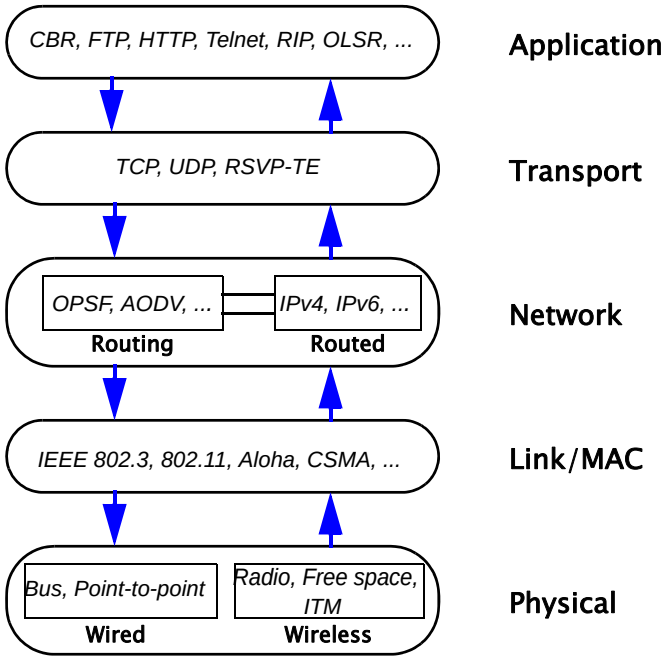


FIGURE 4-1. EXata Protocol Stack

In Section 4.1, we describe the functions used to read configuration parameters. In Section 4.2 through Section 4.6, we describe the procedure to add a custom protocol to each of the layers of the EXata protocol stack. In Section 4.7 and Section 4.8, we describe how to add communication medium models and node mobility models, respectively. In Section 4.9, we describe the procedure to add trace collection to a protocol. In Section 4.10, we describe the procedure to add a custom add-on module. In Section 4.11, we describe the procedure to enable non-adjacent layers to communicate.

4.1 General Programming Utility Functions

4.1.1 Reading Input from a Configuration File

The EXata configuration file is used to configure the protocol stack at each node and to specify the parameters for each protocol. The default configuration file is EXATA_HOME/scenarios/default/default.config.

Protocol parameters are specified using the following format:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where:

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

The types of variables that require instance identifiers are typically arrays of values. As an example, consider the case of priority queues. In the default configuration each node has three priority queues on each interface. The following is an example of specifying weights of the interface queues in the configuration file:

```
QUEUE-WEIGHT[0] 0.5
QUEUE-WEIGHT[1] 0.3
QUEUE-WEIGHT[2] 0.2
```

The value specified for a variable in the configuration file can take several forms: string, integer, double, float, and clocktype. EXata provides API functions for reading each variable format from the configuration file. Prototypes for I/O API functions are specified in EXATA_HOME/include/fileio.h. Some example I/O API functions are listed below. See the file fileio.h or *API Reference Guide* for a complete list of API functions and their parameters.

1. IO_ReadString: This function is used to read a string value when a qualifier (identifier) is specified. IO_ReadString is an overloaded function. One of the versions of this function is described here.

```
void
IO_ReadString(
    const NodeAddress nodeId,
    const NodeAddress interfaceAddress,
    const NodeInput *nodeInput,
    const char *index,
    BOOL *wasFound,
    char *readVal);
```

   The node identifier (`nodeId`), node address (`interfaceAddress`), a pointer to the data representation of the input file (`nodeInput`) and the variable name (`index`) are passed to the function. If a match is found in the input file for the node identifier, node address and variable name, the function sets the Boolean variable `wasFound` to TRUE and sets the destination string pointer (`readVal`) to the string corresponding to the parameter value; if a match is not found, the function sets `wasFound` to FALSE.

2. IO_ReadStringInstance: This function is used to read a string value when both a qualifier and an instance are specified. IO_ReadStringInstance is an overloaded function. One of the versions of this function is described here.

```
void
IO_ReadStringInstance(
    const NodeAddress nodeId,
    const NodeAddress interfaceAddress,
    const NodeInput *nodeInput,
    const char *parameterName,
    const int parameterInstanceNumber,
    const BOOL fallbackIfNoInstanceMatch,
    BOOL *wasFound,
    char *parameterValue);
```

   Function IO_ReadStringIntance is similar to function IO_ReadString, but has two extra parameters. The first, `parameterInstanceNumber`, identifies the parameter instance for which the value is to be read. The other, `fallbackIfNoInstanceMatch`, is a boolean that specifies whether the fallback value for the parameter is assigned to the specified parameter instance if there is no match for `parameterInstanceNumber`.

3. IO_ReadCachedFile: This function is used to read and store the contents of a file when the name of the file is used as a parameter value. IO_ReadCacheFile is an overloaded function. One of the versions of this function is described here.

```
void
IO_ReadCachedFile(
    const NodeAddress nodeId,
    const NodeAddress interfaceAddress,
    const NodeInput *nodeInput,
    const char *parameterName,
    BOOL *wasFound,
    NodeInput *parameterValue);
```

Function IO_ReadCachedFile is similar to function IO_ReadString, except that the contents of the file specified as the parameter value are stored in `parameterValue`. `parameterValue` can then be passed as a parameter to other IO read functions to extract numeric and string values from it.

4.1.2 Programming with Message Info Fields

The message data structure contains fields, called *info* fields, which are used to store extra information used in the processing of the message as well as information that needs to be transported between layers. This information is typically used only in simulation (i.e., it does not have a counterpart in real networks) and, therefore, is not included in the packet payload. This information can be used for simulation tasks, such as collecting statistics and exchanging information across layers or nodes.

The info field is a memory storage associated with a message. Any type of information can be stored in this field. A message can have multiple info fields, each of which is identified by an info field type. Each message contains an info field of the default info field type which can be used by all models. To have exclusive use of an info field, a protocol model can define a new info field type and create, access, and modify info fields of that type.

All info fields are freed automatically when the message is freed. Once a message is freed, its info fields are not accessible anymore.

This section describes the APIs for manipulating info fields which should be sufficient for developing most protocol models. The implementation of the info fields and the advanced info field APIs used for special purposes are not discussed.

4.1.2.1 Info Field Type

The info field types are defined as items of the enumeration `MessageInfoType` in EXATA_HOME/include/message.h (see Figure 4-2).

```
typedef enum message_info_type_str
{
    INFO_TYPE_UNDEFINED = 0,  // an empty info field.
    INFO_TYPE_DEFAULT = 1,    // default info type used in situations where
                              // specific type is given to the info field.
    INFO_TYPE_AbstractCFPropagation, // type for abstract contention free
                                     // propagation info field.
    INFO_TYPE_AppName,      // Pass the App name down to IP layer
    INFO_TYPE_StatCategoryName,
    INFO_TYPE_DscpName,
    ...
    INFO_TYPE_ForwardTcpHeader,
} MessageInfoType;
```

FIGURE 4-2. Info Field Types

4.1.2.2 APIs for Info Field Operations

Several API functions are available in EXata for operations on the info field. These APIs can be called from any layer. The prototypes for these functions can be found in the file message.h. The implementation code for these functions can be found in the file EXATA_HOME/main/message.cpp. Refer to *API Reference Guide* or the file message.h for a more detailed description of these APIs and their parameters.

- MESSAGE_AddInfo: This function creates an info field of the type and size that are passed as parameters. If an info field of the same type already exists in the message, it is replaced with the new info field with the new size. The function returns a pointer to the allocated space, which can be used to access the memory allocated to this info field.
- MESSAGE_InfoAlloc: This function assigns an info field of type `INFO_TYPE_DEFAULT`. The size of the field is passed as a parameter. It is the same as calling MESSAGE_AddInfo with `INFO_TYPE_DEFAULT` as the type.
- MESSAGE_RemoveInfo: This function removes the info field with the specified type from the message. The memory allocated for the info field is also freed. If an info field with the specified type does not exist, no action is taken.
- MESSAGE_ReturnInfo: This is an overloaded function with the following variants:
  - When info field type is passed as a parameter, the function returns a pointer to the info field with the specified type. This pointer can be used to access the space allocated for the info field.
  - When the info field type is not specified as a parameter, the function returns a pointer to the info field with type `INFO_TYPE_DEFAULT`.
- MESSAGE_CopyInfo: This function copies all info fields from one message to another.
- MESSAGE_ReturnInfoSize: This is an overloaded function with the following variants:
  - When the info field type is passed as a parameter, the function returns the size of the info field with the specified type.
  - When the info field type is not passed as a parameter, the function returns the size of the info field with type `INFO_TYPE_DEFAULT`.

4.1.2.3 Using Info Fields

This section describes how to use the info fields in writing protocols.

4.1.2.3.1 Declaring User-defined Info Field Type

The info field type can be one of the items of the enumeration `MessageInfoType`, which is declared in message.h. Users can define their own info field type by including it in the enumeration `MessageInfoType`, as shown in Figure 4-3.

User-defined info field types are particularly useful if the user wants to ensure that other models do not inadvertently modify or delete the data stored in the info field by the user's own model.

Note **Always add to the end of lists in header files. EXata's pre-built object files use the values which existed when the object files were created. Inserting the constant in the middle of the list will result in the values below being offset in any new object files and may lead to the simulator crashing.**

```
typedef enum message_info_type_str
{
    INFO_TYPE_UNDEFINED = 0,  // an empty info field.
    INFO_TYPE_DEFAULT = 1,    // default info type used in situations where
                              // specific type is given to the info field.
    INFO_TYPE_AbstractCFPropagation, // type for abstract contention free
                                     // propagation info field.
    INFO_TYPE_AppName,      // Pass the App name down to IP layer
    INFO_TYPE_StatCategoryName,
    INFO_TYPE_DscpName,
    ...
    INFO_TYPE_ForwardTcpHeader,
    INFO_TYPE_MYINFOTYPE     // Type for Myinfo field
} MessageInfoType;
```

FIGURE 4-3. Declaring User-defined Info Field Type

4.1.2.3.2 Adding an Info Field

The API MESSAGE_AddInfo is used to allocate space for an info field and adding the pointer to the allocated space to the message data structure. Following that, data can be stored in the info field by accessing the pointer to the allocated space.

The following sample code shows how to add an info field of the user-defined type `INFO_TYPE_MYINFO`. It assumes that the structure `MyInfoField` has been defined in the header file of the user model.

```
...
Message* msg;
struct MyInfoField* infoPtr;
...
msg = MESSAGE_Alloc(node, layer, protocol, eventType);

infoPtr = MESSAGE_AddInfo(node,
msg,
sizeof(MyInfoField),
INFO_TYPE_MYINFO);
...
// fill data in the info field using infoPtr now.
...
```

4.1.2.3.3 Accessing an Info Field

The API MESSAGE_ReturnInfo is used to access an info field of a specific type. It returns a pointer which can be used to access data stored in the info field.

The following code sample shows how to access an info field of the user-defined type `INFO_TYPE_MYINFO`.

```
...
struct MyInfoField* infoPtr;
...
infoPtr = MESSAGE_ReturnInfo(msg, INFO_TYPE_MYINFO);
...
// Access fields of MyInfoField using pointer infoPtr.
...
```

4.1.2.3.4 Removing an Info Field

The API MESSAGE_RemoveInfo is used to free the space allocated to an info field of a specific type. Note that the space allocated to a specific info field can be over-written, but it is persistent unless explicitly freed by freeing the specific info field or by freeing the entire message using the API MESSAGE_Free.

The following code sample shows how to remove an info field of the user-defined type `INFO_TYPE_MYINFO`.

```
...
struct MyInfoField* infoPtr;
...
infoPtr = MESSAGE_ReturnInfo(msg, INFO_TYPE_MYINFO);
if (infoPtr != NULL)
{
   ...
   // Access fields of MyInfoField using pointer infoPtr.
   ...
   MESSAGE_RemoveInfo(node, msg, INFO_TYPE_MYINFO);
}
```

4.1.2.4 Persistence of Info Fields

Since the message data structure can have multiple info fields of different types, each model can use an info field to store information relevant to that model. In general, a model only manipulates its own info field and does not modify the other info fields. Therefore, a model's info field is expected to be *persistent*, i.e., it will not be modified by other models. To ensure that a user-created model's info field is persistent, the model should define and use its own info field, as described in Section 4.1.2.3.

Note **To ensure that info fields used by other models are persistent, a user-created model should not modify any info field other than the ones defined by the model itself.**

Note **The default info field is used by many models and should not be assumed to be persistent.**

4.1.3 Random Number Generation

EXata uses sequences of pseudo-random numbers to model a number of real world systems. Random numbers must be used properly to ensure both the accuracy and repeatability of results.

A single sequence of random numbers is referred to as a random stream. A scenario may use many different random streams. In general, an independent random stream is required for each application and some applications may require multiple independent random streams. For example, if a scenario has ten Poisson processes that generate traffic, then it should use ten independent random streams to generate the inter-arrival intervals for each traffic stream. If an application generates traffic where both packet size and inter-packet interval are random, then it should use one random stream for the size and another one for the interval.

Using the same random number stream for multiple purposes leads the generated values to be correlated, and having correlated input streams may have subtle effects on the simulation results or even render the results invalid.

Random streams must be repeatable as well as independent. Given the same starting point, i.e., the same simulation parameters, they should generate exactly the same results; otherwise, it becomes impossible to verify the behavior of the system.

This section explains how EXata's random number generation package is organized and gives examples of its usage.

4.1.3.1 Basic Functions for Random Number Generation

EXata's random number generation system is built around three basic functions: RANDOM_erand, RANDOM_jrand, and RANDOM_nrand. These functions are based on the 48-bit random number generators, erand48, jrand48, and nrand48, found on most UNIX systems. File EXATA_HOME/include/random.h contains the prototypes of these functions and other declarations related to random number generation.

The following declaration defines the type for the 48-bit seed used by the random number generators:

```
typedef unsigned short RandomSeed[3];
```

The random generator functions are:

- `extern double RANDOM_erand(RandomSeed);`
  This function returns a real number between 0.0 and 1.0, both inclusive.
- `extern Int32 RANDOM_jrand(RandomSeed);`
  This function returns an integer between -2^{31} and 2^{31}, both inclusive.
- `extern Int32 RANDOM_nrand(RandomSeed);`
  This function returns an integer between 0 and 2^{31}, both inclusive.

Each call to RANDOM_erand, RANDOM_jrand, or RANDOM_nrand generates a random number based on the seed that is passed as a parameter and updates the seed. The next call to the function uses the updated seed to generate a new random number and updates the seed again. The updates to the seed are deterministic and the entire stream of generated random numbers is determined by the initial seed. In order to create two independent random streams, two seed variables with independently assigned initial values are required.

The following function is used to set the initial seed:

```
void RANDOM_SetSeed(RandomSeed seed,
                    UInt32 globalSeed,
                    UInt32 nodeId = 0,
                    UInt32 protocolId = 0,
                    UInt32 instanceId = 0);
```

The input parameters of this function are used to generate deterministic, but unique initial seeds:

- `globalSeed`: This is the `SEED` parameter in the configuration file. Including this parameter allows the user to change the random stream for different experiments.
- `nodeId`: Including the node identifier ensures that each node will use a different random stream.
- `protocolId`: Including the protocol identifier ensures that different protocols at the same node use different random streams.
- `instanceId`: Including the instance identifier ensures that different instances of the same protocol use different random streams.

We illustrate the use of the basic random number generator functions by taking as an example the EXata implementation of the Multiple Access Collision Avoidance (MACA) MAC protocol. MACA implementation code is contained in the files mac_maca.h and mac_maca.cpp in the folder EXATA_HOME/libraries/wireless/src.

The MACA protocol uses two random streams: one to generate random backoff times and the other to generate random channel yield times. Both random streams are uniformly distributed. The MACA data structure, `MacDataMaca`, includes two variables, `backoffSeed` and `yieldSeed`, of type `RandomSeed`, to store the seeds for the two distributions. These seed variables are initialized in the function MacMacaInit, as shown in Figure 4-4.

```
void MacMacaInit(
   Node *node, int interfaceIndex, const NodeInput *nodeInput)
{
    MacDataMaca *maca = (MacDataMaca *) MEM_malloc(sizeof(MacDataMaca));

    assert(maca != NULL);

    memset(maca, 0, sizeof(MacDataMaca));
    maca->myMacData = node->macData[interfaceIndex];
    maca->myMacData->macVar = (void *)maca;
    ...
    maca->currentNextHopAddress = ANY_DEST;

    RANDOM_SetSeed(maca->backoffSeed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_MACA,
                   interfaceIndex);
    RANDOM_SetSeed(maca->yieldSeed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_MACA,
                   interfaceIndex + 1);
    ...
}
```

FIGURE 4-4. Setting Random Number Seeds

Note that `backoffSeed` is set by calling RANDOM_SetSeed with `interfaceIndex` as the last parameter, whereas `yieldSeed` is set by calling RANDOM_SetSeed with `interfaceIndex + 1` as the last parameter. This ensures that different random streams will be used for backoff and yield times.

Once the two independent seeds have been set, RANDOM_erand, RANDOM_jrand, and RANDOM_nrand can be used to get the next random number in the sequence by passing the proper seed as the parameter. Figure 4-5 shows function MacMacaYield which calls function RANDOM_nrand with `maca->yieldSeed` as the parameter to get a random yield time from a uniform distribution. Figure 4-6 shows function MacMacaBackoff which calls function RANDOM_nrand with `maca->backoffSeed` as the parameter to get a random backoff time from a uniform distribution.

```
static
void MacMacaYield(Node *node, MacDataMaca *maca, clocktype vacation)
{
    assert(maca->state == MACA_S_YIELD);
    MacMacaSetTimer(node, maca, MACA_T_YIELD,
                    vacation + RANDOM_nrand(maca->yieldSeed) % 20);
}
```

FIGURE 4-5. Generating a Random Value for Yield Time

```
static
void MacMacaBackoff(Node *node, MacDataMaca *maca)
{
    clocktype randomTime;
    assert(maca->state == MACA_S_BACKOFF);
    randomTime = (RANDOM_nrand(maca->backoffSeed) % maca->BOmin) + 1;
    ...
    MacMacaSetTimer(node, maca, MACA_T_BACKOFF, randomTime);
}
```

FIGURE 4-6. Generating a Random Value for Backoff Time

4.1.3.2 Built-in Random Number Distributions

The functions described in Section 4.1.3.1 can be used to generate random numbers and transform them into the type of distribution required. EXata also provides several built-in distributions that can be used directly. This section describes the interface to the built-in distributions.

EXata random number distributions are implemented by means of a C++ class, `RandomDistribution`, which is defined in random.h and is shown in Figure 4-7.

```
template <class T>
class RandomDistribution
{
public:
    ...
    void setDistributionUniform(T min, T max);
    void setDistributionUniformInteger(T min, T max);
    void setDistributionExponential(T mean);
    void setDistributionGaussian(double sigma);
    void setDistributionGaussianInt(double sigma);
    void setDistributionPareto(T val1, T val2, double alpha);
    void setDistributionPareto4(T val1, T val2, T val3, double alpha);
    void setDistributionGeneralPareto(T val1, double alpha);
    void setDistributionParetoUntruncated(T val1, double alpha);
    void setDistributionDeterministic(T val);
    void setDistributionNull();
    int setDistribution(char* inputString,
                        char* printStr,
                        RandomDataType dataType);
    T getRandomNumber();
    T getRandomNumber(RandomSeed seed);
    void setSeed(UInt32 globalSeed,
                 UInt32 nodeId = 0,
                 UInt32 protocolId = 0,
                 UInt32 instanceId = 0);
 ...
};
```

FIGURE 4-7. Class `RandomDistribution`

The RandomDistribution class has four types of member functions that are of interest to programmers:

- **Set Distribution Functions**: These functions set the distribution type.
  - setDistributionUniform: This function sets the distribution to return a value x, where x is uniformly distributed in the range min <= x < max.
  - setDistributionUniformInteger: This function sets the distribution to return a value x, where x is uniformly distributed in the range min <= x <= max.

    Note: This function should only be used for integer variables.
  - setDistributionExponential: This function sets an exponential distribution with mean as the mean.
  - setDistributionGaussian: This function sets a Gaussian distribution with sigma as the sigma value.
  - setDistributionGaussianInteger: This function sets a Gaussian distribution with sigma as the sigma value, but returns only integers.
  - setDistributionGeneralPareto: This function sets a Generalized Pareto distribution with val1 as the lower end of the range and alpha as the shape parameter.
  - setDistributionParetoUntruncated: This function sets an untruncated Pareto distribution with val1 as the lower end of the range and alpha as the shape parameter.
  - setDistributionPareto4: This function sets a truncated Pareto distribution with val1 as the lower end of the range, val2 as the lower limit of the truncation, val3 as the upper limit of the truncation, and alpha as the shape parameter.
  - setDistributionPareto: This function sets a truncated Pareto distribution with val1 as the lower end of the range (= the lower limit of the truncation), val2 as the upper limit of the truncation, and alpha as the shape parameter.
  - setDistributionDeterministic: This function sets the distribution to always return val.
- **File Parsing Function**: Function setDistribution parses an input string, determines the type of distribution and the parameters associated with it, and calls the set distribution function for that distribution. This is explained in detail in Section 4.1.3.2.2.
- **Get Random Number Function**: Function getRandomNumber returns the next random number in the sequence according to the distribution that was set by calling one of the set distribution functions.
- **Set Seed Function**: Function setSeed sets the initial seed for the random sequence. This function is similar to the function RANDOM_SetSeed described in Section 4.1.3.1.

Section 4.1.3.2.1 and Section 4.1.3.2.2 give examples of using the RandomDistribution class. The example in Section 4.1.3.2.2 also illustrates the use of the file parsing function.

4.1.3.2.1 Using the RandomDistribution Class

We illustrate the use of the built-in random number distributions by taking as an example EXata modeling of shadowing effects.

The data structure for storing propagation data, PropData, shown in Figure 4-8, contains a variable shadowingDistribution of type RandomDistribution.

```
struct PropData {
    int   numPhysListenable;
    int   numPhysListening;
    ...
    RandomDistribution<double> shadowingDistribution;
    int  nodeListId;
    int  numSignals;
    ...
};
```

FIGURE 4-8. Declaring a Random Distribution Variable

Initializing a distribution comprises two steps: setting the initial seed and setting the distribution type. EXata implements two types of shadowing models: constant and lognormal. If the constant shadowing model is specified, then the deterministic distribution is used. If the lognormal shadowing model is specified, then the Gaussian distribution is used. Function PROP_Init, shown in Figure 4-9, sets the initial seed for `shadowingDistribution` and sets `shadowingDistribution` to deterministic or Gaussian depending on the shadowing model specified. PROP_Init is implemented in EXATA_HOME/libraries/wireless/src/propagation.cpp.

```
void PROP_Init(Node *node, int channelIndex, NodeInput *nodeInput) {
    PropData* propData = &(node->propData[channelIndex]);
    ...
    propData->shadowingDistribution.setSeed(
        node->globalSeed,
        node->nodeId,
        channelIndex);
    if (propProfile->shadowingModel == CONSTANT) {
        propData->shadowingDistribution.setDistributionDeterministic(
            propProfile->shadowingMean_dB);
    }
    else { // propProfile->shadowingModel == LOGNORMAL
        propData->shadowingDistribution.setDistributionGaussian(
            propProfile->shadowingMean_dB);
    }
```

FIGURE 4-9. Initializing a Random Distribution

Function PROP_CalculatePathloss, shown in Figure 4-10, calls shadowingDistribution.getRandomNumber to obtain a number from the random distribution. The random number that is returned is generated according to the distribution that was set in PROP_Init. PROP_CalculatePathloss is implemented in propgataion.cpp.

```
void PROP_CalculatePathloss(
    Node* node,
    int channelIndex,
    double wavelength,
    float txAntennaHeight,
    float rxAntennaHeight,
    PropPathProfile *pathProfile,
    double* pathloss_dB)
{
    ...
    switch (propProfile->pathlossModel) {
        case FREE_SPACE:
        case TWO_RAY:
        {
            double shadowing_dB = 0.0;
            if (propProfile->shadowingMean_dB != 0.0) {
                shadowing_dB =
                  propData->shadowingDistribution.getRandomNumber();
            }
            ...
            return;
          }
         ...
    }
    return;
}
```

FIGURE 4-10. Acquiring Numbers from a Random Distribution

4.1.3.2.2 Using the File Parsing Function

If the distribution to be used for a random variable is known a-priori, then the built-in distributions can be used by calling the set seed and appropriate set distribution functions, as described in Section 4.1.3.2.1. However, in some cases, the distribution may be specified by the user and the model may need to read it from a file (typically, the .app file). The `RandomDistribution` class implements a function, setDistrbution, which parses an input line to determine the distribution type and its associated parameters and initializes the appropriate distribution.

A line in the input file may have specifications for one or more random distributions. Each random distribution is specified in the following format:

```
<Distribution Identifier> <Parameter List>
```

where

`<Distribution Identifier>` : String identifying the distribution.

`<Parameter List>` : Parameters for the distribution.

The string identifier and parameters for random distributions that can be read from an input file are listed in Table 4-1.

TABLE 4-1. Distribution Identifiers and Parameters

Distribution Name	Distribution Identifier	Parameters
Uniform	`UNI`	• Lower end of the range • Upper end of the range
Exponential	`EXP`	• Mean value
Pareto	`TPD`	• Lower end of the range (= lower limit of the truncation) • Upper limit of the truncation • Shape parameter
Pareto4	`TPD4`	• Lower end of the range • Lower limit of the truncation • Upper limit of the truncation • Shape parameter
Deterministic	`DET`	• Value

Examples:

`UNI 10 30` : Denotes a uniform distribution in the range 10 to 30

`DET 20MS` : Denotes a deterministic distribution with the value 20 milliseconds.

Function setDistribution has three input parameters and returns an integer value. The input parameters are:

- Input string: String that has to be parsed.
- Print string: String used for printing error messages (typically, the protocol name).
- Data type: Indication of the type of value the distribution is to return. It can be one of the values (`RANDOM_INT`, `RANDOM_DOUBLE` or `RANDOM_CLOCKTYPE`) of the enumeration `RandomDataType` defined in random.h. It is used to convert numeric parameters read from the input string into the correct data type.

The integer value returned by the function setDistribution is the number of tokens in the input string required for specifying the distribution, i.e., the number of parameters associated with the distribution (see Table 4-1) plus 1.

We illustrate the use of the file parsing utility, setDistribution, by using the implementation of the Traffic Generator application (TRAFFFIC-GEN) as an example. TRAFFIC-GEN is implemented by files app_traffic_gen.h and app_traffic_gen.cpp in the folder EXATA_HOME/libraries/developer/src.

TRAFFIC-GEN uses random distribution for the packet size, packet interval, and drop probability. The drop probability is modeled by a uniform distribution in the range (0.0, 1.0). The packet size and packet interval distributions can be configured by the user and specified in the .app file.

The distributions are implemented by three variables of `RandomDistribution` type which are part of the data structure for the TRAFFIC-GEN client, `TrafficGenClient`, shown in Figure 4-11 and declared in file app_traffic_gen.h.

```
typedef struct struct_traffic_gen_client
{
    // Two end nodes
    NodeAddress     localAddr;
    NodeAddress     remoteAddr;
    ...
    // Random dist. traffic properties
    RandomDistribution<UInt32>    dataSizeDistribution;
                                        // Data length traffic gen dist.
    RandomDistribution<clocktype> intervalDistribution;
                                        // Data interval traffic gen dist.
    RandomDistribution<double>    probabilityDistribution;
                                        // general probability distribution.
    double          genProb;            // Data generation probability
    ...
} TrafficGenClient;
```

FIGURE 4-11. Declaring Distribution Variables

The distributions are initialized in the functions TrafficGenClientNewClient and TrafficGenClientInit, which are both implemented in the file app_traffic_gen.cpp. Since the packet size and interval distribution types are read from the input file, the `dataSizeDistribution` and `intervalDistribution` distributions are initialized to null in function TrafficGenClientNewClient, as shown in Figure 4-12.

```
static
TrafficGenClient* TrafficGenClientNewClient(Node* node)
{
    TrafficGenClient* clientPtr = (TrafficGenClient*)
            MEM_malloc(sizeof(TrafficGenClient));
    // Initialize the client
    ...
    clientPtr->dataSizeDistribution.setDistributionNull();
    clientPtr->intervalDistribution.setDistributionNull();
    ...
 }
```

FIGURE 4-12. Initializing Distribution Variables: Part 1

In function TrafficGenClientInit (see Figure 4-13), independent, unique seeds for the three distributions (packet size, packet interval, and drop probability) are set by calling the `RandomDistribution` function setSeed.

`probabilityDistribution` is set to be a uniform distribution by calling function probabilityDistribution.setDistributionUniform.

Function dataSizeDistribution.setDistribution scans the input string (`tokenStr`). The first token encountered is the distribution identifier, which determines how many parameters follow the distribution identifier. Function dataSizeDistribution.setDistribution reads the appropriate number of tokens from the input string, converts the numeric data to the proper type (`int`, in this case) and calls the set distribution function corresponding to the distribution identifier. The number of tokens read (3, in this case) is returned and assigned to `nToken`.

Function TrafficGenClientSkipToken skips `nToken` number of tokens in the input string (`tokenStr`).

Function intervalDistribution.setDistribution behaves in the same way as function dataSizeDistribution.setDistribution, except that in this case, the numeric data are converted to type `clocktype`.

```
void TrafficGenClientInit(
    Node* node,
    char* inputString,
    NodeAddress localAddr,
    NodeAddress remoteAddr,
    DestinationType destType)
{
    char buf[MAX_STRING_LENGTH];
    TrafficGenClient* clientPtr = NULL;
    char* tokenStr = NULL;
    int nToken;
    ...
    // Initialize each distribution with a different seed for independence.
    ...
    clientPtr->dataSizeDistribution.setSeed(node->globalSeed,
                                            node->nodeId,
                                            APP_TRAFFIC_GEN_CLIENT,
                                            2);
    clientPtr->intervalDistribution.setSeed(node->globalSeed,
                                            node->nodeId,
                                            APP_TRAFFIC_GEN_CLIENT,
                                            3);
    clientPtr->probabilityDistribution.setSeed(node->globalSeed,
                                               node->nodeId,
                                               APP_TRAFFIC_GEN_CLIENT,
                                               4);
    clientPtr->probabilityDistribution.setDistributionUniform(0.0, 1.0);
    ...
    if (strcmp(buf, "RND") == 0)
    {
        // Random distribution traffic
        clientPtr->trfType = TRAFFIC_GEN_TRF_TYPE_RND;
        nToken = clientPtr->dataSizeDistribution.setDistribution(tokenStr,
                                                                 "TrafficGen",
                                                                 RANDOM_INT);
        tokenStr = TrafficGenClientSkipToken(tokenStr, TOKENSEP, nToken);
        nToken = clientPtr->intervalDistribution.setDistribution(tokenStr,
                                                                 "TrafficGen",
                                                            RANDOM_CLOCKTYPE);
        tokenStr = TrafficGenClientSkipToken(tokenStr, TOKENSEP, nToken);
        ...
    }
    else if (strcmp(buf, "TRC") == 0)
    {
        ...
    }
    else
    {
        ...
    }
    ...
}
```

FIGURE 4-13. Initializing Distribution Variables: Part 2

Example:

Consider the following input string:

```
TRAFFIC-GEN 1 11 DET 180 DET 900 RND UNI 200 250 UNI 20US 20MS 1 NOLB
```

The first eight tokens (`TRAFFIC-GEN`, `1`, `11`, `DET`, `180`, `DET`, `900`, and `RND`) are processed by code not shown in Figure 4-13.

Call to function dataSizeDistribution.setDistribution reads the distribution identifier "`UNI`". Since that indicates a uniform distribution, the next two tokens are read as the parameters of the uniform distribution. These parameters are converted to `int` type since the last parameter to function dataSizeDistribution.setDistribution is `RANDOM_INT`. Function dataSizeDistribution.setDistribution also sets the distribution `dataSizeDistribution` to be a uniform distribution with the range 200 to 250.

Next, function TrafficGenClientSkipToken skips three tokens in `tokenStr`.

Function intervalDistribution.getDistribution is similar to function dataSizeDistribution.setDistribution. It reads the next three tokens and sets `intervalDistribution` to be a uniform distribution that returns a `clocktype` value in the range 20 microseconds to 20 milliseconds.

After initialization, the random distributions can be used by calling the appropriate getRandomNumber function. Figure 4-14 shows how this is done in function TrafficGenClientInit.

```
void TrafficGenClientInit(
    Node* node,
    char* inputString,
    NodeAddress localAddr,
    NodeAddress remoteAddr,
    DestinationType destType)
{
    char buf[MAX_STRING_LENGTH];
    TrafficGenClient* clientPtr = NULL;
    char* tokenStr = NULL;
    int nToken;
    ...
    if (strcmp(buf, "CONSTRAINT") == 0)
        {
            ...
            unsigned int dataLen = (unsigned int)
                clientPtr->dataSizeDistribution.getRandomNumber();
            ...
            clocktype dataIntv;
            unsigned int sessionBwRequirement;
            dataIntv = clientPtr->intervalDistribution.getRandomNumber();
            ...
        }
      ...
}
```

FIGURE 4-14. Acquiring Numbers from Random Distributions

4.2 Application Layer

The Application Layer is the topmost layer in the protocol stack, as shown in Figure 4-1. User applications and some routing protocols reside at this layer.

This section gives a detailed description of how to add an Application Layer protocol to EXata.

4.2.1 Application Layer Protocols in EXata

EXata provides a large number of Application Layer protocols. Multiple applications, and multiple instances of the same application, can run at a node simultaneously, much like a real network.

Application Layer protocols in EXata can be grouped into the following two categories:

- Traffic-generating Protocols
- Routing Protocols

4.2.1.1 Traffic-generating Protocols

Traffic-generating protocols simulate the traffic generated by a real network application. EXata provides a large number of traffic-generating protocols. Table 4-2 lists some of the traffic-generating Application Layer protocols in EXata.

While some protocols are used directly as applications, such as FTP and Telnet, others are used to simulate real network applications. Applications such as CBR (Constant Bit Rate) can be configured to simulate a large number of real network applications by mimicking their traffic pattern. For example, audio traffic and old video codecs infuse traffic at a constant rate into the network and can be accurately simulated by appropriately configuring the CBR application in EXata.

Table 4-2 lists the different traffic generators modeled in EXata. See the corresponding model library for a detailed description of each protocol and its parameters.

TABLE 4-2. Traffic Generators in EXata

Traffic Generator	Description	Model Library
CBR	Constant Bit Rate (CBR) traffic generator. This UDP-based client-server application sends data from a client to a server at a constant bit rate.	Developer
CELLULAR-ABSTRACT-APP	Abstract cellular application. This is an application to generate traffic for networks running abstract cellular models.	Cellular
FTP	File Transfer Protocol (FTP). This tcplib application generates TCP traffic based on historical trace data.	Developer
FTP/GENERIC	Generic FTP. This model is similar to the FTP model but allows the user to have more control over the traffic properties. It uses FTP to transfer a user-specified amount of data.	Developer
GSM	Global System for Mobile communications (GSM). This is an application for generating traffic for GSM networks.	Cellular

TABLE 4-2. Traffic Generators in EXata (Continued)

Traffic Generator	Description	Model Library
HTTP	HyperText Transfer Protocol (HTTP). The HTTP application generates realistic web traffic between a client and one or more servers. The traffic is randomly generated based on historical data.	Developer
LOOKUP	Look-up traffic generator. This is an abstract model of unreliable query/response traffic, such as DNS look-up, or pinging.	Developer
MCBR	Multicast Constant Bit rate (MCBR). This model is similar to CBR and generates multicast constant bit rate traffic.	Developer
PHONE-CALL	Phone call traffic generator. This model simulates phone calls between two end users in a UMTS network.	UMTS
SUPER-APPLICATION	Super application. This model can simulate both TCP and UDP flows as well as two-way (request-response type) UDP sessions.	Developer
TELNET	Telnet application. This model generates realistic Tenet-style TCP traffic between a client and a server based on historical data. It is part of the tcplib suit of applications.	Developer
TRAFFIC-GEN	Random distribution-based traffic generator. This is a flexible UDP traffic generator that supports a variety of data size and interval distributions and QoS parameters.	Developer
TRAFFIC-TRACE	Trace file-based traffic generator. This model generates traffic according to a user-specified file, and like Traffic-Gen, it supports QoS parameters.	Developer
VBR	Variable Bit Rate (VBR) traffic generator. This model generates fixed-size data packets transmitted using UDP at exponentially distributed time intervals.	Developer
VOIP	Voice over IP traffic generator. This model simulates IP telephony sessions.	Multimedia and Enterprise
ZIGBEEAPP	ZigBee Application. This is similar to the CBR application but is used only in sensor networks.	Sensor Networks

4.2.1.2 Routing Protocols

In addition to traffic generators, certain service-providing protocols may also reside at the Application Layer. Routing protocols is a common category of service-providing Application Layer protocols. These routing protocols use UDP or TCP services.

Table 4-3 lists the Application Layer routing protocols in EXata. See the corresponding model library for a detailed description of each protocol and its parameters.

Note **Some routing protocols are implemented at the Network Layer (see Table 4-8 and Table 4-9).**

TABLE 4-3. Application Layer Routing Protocols in EXata

Routing Protocol	Description	Model Library
BELLMANFORD	Bellman-Ford routing protocol.	Developer
BGPv4	Border Gateway Protocol version 4 (BGPv4). This protocol can be used for IPv4 and IPv6 networks.	Multimedia and Enterprise
EIGRP	Enhanced Interior Gateway Routing Protocol (EIGRP). This is a distance vector routing protocol designed for fast convergence.	Multimedia and Enterprise
FISHEYE	Fisheye Routing Protocol. This is a link state-based routing protocol.	Wireless
IGRP	Interior Gateway Routing Protocol (IGRP). This is a distance vector Interior Gateway protocol (IGP).	Multimedia and Enterprise
OLSR-INRIA	Optimized Link State Routing (OLSR) protocol. This is a link state-based routing protocol.	Wireless
OLSRv2-NIIGATA	Optimized Link State Routing, version 2 (OLSRv2) protocol. This is a successor of the OLSR protocol.	Wireless
RIP	Routing Information Protocol (RIP) routing protocol.	Developer
RIPng	Routing Information Protocol, next generation (RIPng) routing protocol. This protocol can be used for IPv6 networks.	Developer

Other routing protocols may send messages directly from the Network Layer. These protocols do not use UDP or TCP services. Examples of such Network Layer routing protocols include the Ad-hoc Distance Vector (AODV) and Dynamic Source Routing (DSR) protocols. Table 4-4 lists some differences between Application Layer and Network Layer routing protocols

TABLE 4-4. Application Layer versus Network Layer Routing Protocols

Application Layer Routing Protocols	Network Layer Routing Protocols
Use UDP or TCP to transmit their route discovery and control packets.	Use IP directly to transmit their route discovery and control packets.
Use an IP kernel function to update the IP forwarding table.	Use IP kernel functions to register itself as the packet routing function.
Do not receive data packets to forward, IP handles those itself.	Receive data packets and decide outgoing interface to forward packets.

4.2.2 Application Layer Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to Application Layer protocols. These files contain detailed comments on functions and other code components.

The Application Layer API is composed of several macros, functions, and structures. These are defined in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/application.h

  This file contains definitions common to Application Layer protocols and Application Layer data structure in the node structure.
- EXATA_HOME/include /app_util.h

  This file contains prototypes of the functions defined in the file EXATA_HOME/main/app_util.cpp.

Additionally, the following header files are also relevant to the Application Layer:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.
- EXATA_HOME/include/mapping.h

  This file contains prototypes of functions to map between node ids and IP addresses.

The following are the folders and source files associated with the Application Layer:

- EXATA_HOME/libraries/developer/src

  This folder contains the source and header files for most of the applications implemented in EXata. The file names are based on the name of the application that they implement, e.g., to see the implementation for CBR (Constant Bit Rate), look at files app_cbr.cpp and app_cbr.h in this folder. Other libraries may contain code for application models as well.
- EXATA_HOME/main/application.cpp

  This file contains Application Layer functions, including the initialization, message processing, and finalization functions.
- EXATA_HOME/main/app_util.cpp

  This file contains utilities used by Application Layer protocols. This includes functions to set timers, register an application, send packets, and manage connections to Transport Layer protocols (UDP and TCP).

4.2.3 Application Layer Data Structures

The Application Layer data structures are defined in EXATA_HOME/include/application.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file application.h for a complete description.)

1. `AppType`: This is an enumeration type that lists all the Application Layer protocols. Note that for each traffic-generating protocols, there are two entries in the list: one for the client and one for the server. There is a single entry in the list for each Application Layer routing protocol.

```
typedef enum
{
    APP_FTP_SERVER_DATA = 20,
    APP_FTP_SERVER = 21,
    APP_FTP_CLIENT,
    APP_TELNET_SERVER = 23,
    APP_TELNET_CLIENT,
    ...
    /* Application-layer routing protocols */
    ...
    APP_ROUTING_FISHEYE = 160,      // IP protocol number
    APP_ROUTING_STATIC,
    ...
    APP_PLACEHOLDER
} AppType;
```

2. `AppInfo`: This data structure contains information about an instance of an application. The information stored is the application type and a pointer to the structure that stores the application state and statistics. Each node maintains this information for each instance of each application running at that node.

```
typedef struct app_info
{
    AppType appType;          /* type of application */
    void *appDetail;          /* statistics of the application */
    struct app_info *appNext; /* link to the next app of
                                 the node */
} AppInfo;
```

3. `AppData`: This is the main data structure used by the Application Layer and stores information about all applications running at a node. Some important fields of this structure are explained below.

```
struct AppData
{
    AppInfo  *appPtr;       /* pointer to the list of app info */
    PortInfo *portTable;    /* pointer to the port table */
    short    nextPortNum;   /* next available port number */
    BOOL     appStats;      /* flag indicating whether application
                               statistics collection is enabled */
    AppType exteriorGatewayProtocol;
    BOOL routingStats;
    void *routingVar;
    void *bellmanford;
    void *olsr;
    ...
};
```

- `appPtr`: This is a pointer to the list of traffic-generating protocols running at the node. Each instance of an application has its own entry in this list.
- `appStats`: This flag indicates whether or not statistics collection is enabled for the Application Layer.
- `portTable`, `nextPortNum`: These fields are used to manage port numbers and are explained in Section 4.2.7.1.
- `routingVar`, `bellmanford`, `olsr`: These are pointers to the Application Layer routing protocols running at the node.

4. `AppTimer`: This data structure is used to implement Application Layer timers. It stores the timer type and information to identify the application for which the timer is set.

```
typedef struct app_timer
{
  int type;               /* timer type */
  int connectionId;       /*  the connection this timer is meant for */
  unsigned short sourcePort;     /*  the port of the session this */
                                 /*  timer is meant for */
   NodeAddress address;  /* address and port combination identify */
                         /* session */
} AppTimer;
```

4.2.4 Application Layer APIs and Inter-layer Communication

This section describes the APIs that are available for the Application Layer to communicate with the Transport Layer (see Section 4.2.4.1), message types that are used by the Transport Layer to communicate with the Application Layer (see Section 4.2.4.2), and some of the Application Layer utility APIs (see Section 4.2.4.3).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.2.4.1 Application Layer to Transport Layer Communication

A number of APIs are available at the Application Layer to communicate with the Transport Layer. The prototypes for the API functions are contained in the file app_util.h. The file app_util.cpp contains the implementation of these functions.

Some of the APIs used for communication from the Application Layer to the Transport Layer are listed below.

- APP_UdpSendNewDataWithPriority: This function allocates and sends data to the UDP protocol at the Transport Layer with the specified priority.
- APP_TcpOpenConnection: This function opens a TCP connection.
- APP_TcpServerListen: This function enables the Application Layer to listen on the specified server port.
- APP_TcpSendData: This function sends data to the TCP protocol at the Transport Layer.
- APP_TcpCloseConnection: This function closes a TCP connection.

4.2.4.2 Transport Layer to Application Layer Communication

Transport Layer protocols communicate with the Application Layer by means of messages. The message types used for this communication are enumerated in the file EXATA_HOME/include/api.h. Some of the message types used by Transport Layer protocols to communicate with the Application Layer are listed below.

- `MSG_APP_FromTransport`: This message type is used by UDP to pass an incoming packet to the Application Layer.
- `MSG_APP_FromTransOpenResult`: This message type is used by TCP to notify an application client that a TCP connection request was accepted or rejected.
- `MSG_APP_FromTransDataSent`: This message type is used by TCP to indicate to the Application Layer that an outgoing packet has been transmitted.
- `MSG_APP_FromTransDataReceived`: This message type is used by TCP to pass an incoming packet to the Application Layer.
- `MSG_APP_FromTransListenResult`: This message type is used by TCP to notify an application server that a request to open a TCP connection has been received.
- `MSG_APP_FromTransCloseResult`: This message type is used by TCP to notify an application client or server that a TCP connection has been closed.

4.2.4.3 Application Layer Utility APIs

Several APIs are available at the Application Layer that perform tasks internal to the Application Layer. The prototypes for the API functions are contained in the file app_util.h. The file app_util.cpp contains the implementation of these functions.

Some of the Application Layer utility APIs are listed below.

- APP_IsFreePort: This functions checks whether the specified port number is free or in use.
- APP_GetProtocolType: This function returns the protocol type for which the specified message is destined.
- APP_RegisterNewApp: This function inserts a new application instance in the list of application instances running at a node's Application Layer.
- APP_SetTimer: This function sets an Application Layer timer.

4.2.5 Adding a Traffic-generating Application Protocol

Although the working of each Application Layer protocol is different, there are certain functions that are performed by most Application Layer protocols. This section provides an overview of the flow of a traffic-generating Application Layer protocol and provides an outline for developing and adding a traffic-generating Application Layer protocol to EXata. It describes how to develop code components common to most application protocols such as initializing, sending and receiving packets, and collecting statistics.

We illustrate the process of adding a traffic-generating protocol by using as an example the implementation code for the CBR (Constant Bit Rate) application, which is one of the most frequently used protocols. The header file for the CBR implementation is app_cbr.h and the source file is app_cbr.cpp in the folder EXATA_HOME/libraries/developer/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a traffic-generating protocol. After understanding the discussed snippets, look at the complete code for CBR to understand how a traffic-generating protocol is implemented in EXata.

CBR, which is used as an example in this section, is a UDP-based application. TCP-based applications, such as FTP, require some additional tasks that are not covered in this section. Use FTP as an example to develop a TCP-based application. The header file for the FTP implementation is app_ftp.h and the source file is app_ftp.cpp in the folder EXATA_HOME/libraries/developer/src.

The following list summarizes the actions that need to be performed for adding a traffic-generating Application Layer protocol to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.2.5.2).
2. Modify the file application.cpp to include the protocol's header file (see Section 4.2.5.2).
3. Include the protocol in the list of Application Layer protocols and trace protocols (see Section 4.2.5.3).
4. Define data structures for the protocol (see Section 4.2.5.4).
5. Decide on the format for the protocol-specific configuration parameters (see Section 4.2.5.5.1).
6. Read the protocol's configuration parameters and call the protocol's initialization function from the Application Layer initialization function, APP_InitializeApplications (see Section 4.2.5.5.2).
7. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Declare and initialize the state variables (see Section 4.2.5.5.3.1).
   b. Register the application instance (see Section 4.2.5.5.3.2).
   c. Initialize timers (see Section 4.2.5.5.3.3).
8. Call the client and server event dispatchers from the Application Layer event dispatcher, APP_ProcessEvent (see Section 4.2.5.6.1).
9. Declare any new event types used by the protocol in the header file api.h (see Section 4.2.5.6.2).
10. Write the client event dispatcher (see Section 4.2.5.6.2).
11. Write the server event dispatcher (see Section 4.2.5.6.3).
12. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.2.5.7.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.2.5.7.2).
    c. Update the statistics as appropriate (see Section 4.2.5.7.3).
    d. Write a function to print the statistics (see Section 4.2.5.7.4).
    e. Add dynamic statistics to the protocol, if desired (see Section 4.2.5.7.5) .
13. Call the client and server finalization function from the Application Layer finalization function, APP_Finalize (see Section 4.2.5.8.1).
14. Write the client finalization function (see Section 4.2.5.8.2). Call the function to print statistics from the client finalization function.
15. Write the server finalization function (see Section 4.2.5.8.3). Call the function to print statistics from the server finalization function.

16. Include the protocol header and source files in the EXata tree and compile (see Section 4.2.5.9).

17. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.2.5.10).

4.2.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new protocol, the programmer name the different components of the new protocol in a similar manner. It will be helpful to examine the implementation of CBR in EXata for hints for naming and coding different components of the new protocol.

In this section, we describe the steps for developing a traffic-generating Application Layer protocol called "MYPROTOCOL". We will use the string "Myprotocol" in the names of the different components of this protocol, just as the string "Cbr" appears in the names of the components of the CBR implementation.

4.2.5.2 Creating Files

The first step towards adding an application model is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/developer/src. However, it is recommended that all user-developed models be made part of a separate library. In our example, we will place the application model in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn't already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the application model and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *app_* to designate the files as application model files.

Examples*:*

- app_ftp.h, app_ftp.cpp: Implement FTP (File Transfer Protocol)
- app_cbr.h, app_cbr.cpp: Implement CBR (Constant Bit Rate)

In keeping with the naming guidelines of Section 4.2.5.1, the header file for the example protocol is called app_myprotocol.h, and the source file is called app_myprotocol.cpp.

It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, app_myprotocol.h, should contain the following:

- Prototypes for interface functions in the source file, app_myprotocol.cpp
- Constant definitions
- Data structure definitions and data types: `struct` and `enum` declarations

The source file, app_myprotocol.cpp, should contain the following:

- Statement to include the protocol's header file:

```
#include "app_myprotocol.h"
```

- Statements to include standard library functions and other header files needed by the protocol source file. A typical protocol source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "api.h"        // EXATA_HOME/include/api.h
#include "app_util.h"   // EXATA_HOME/include/app_util.h
#include "partition.h" // EXATA_HOME/include/partition.h
```

- Initialization functions for the client and server, AppMyprotocolClientInit and AppMyprotocolServerInit, respectively
- Event dispatcher function for the client and server, AppLayerMyprotocolClient and AppLayerMyprotocolServer, respectively
- Finalization function for the client and server, AppMyprotocolClientFinalize and AppMyprotocolServerFinalize, respectively
- Additional protocol implementation functions

The file EXATA_HOME/main/application.cpp contains the layer level initialization, event dispatcher, and finalization functions. These layer level functions in turn call the protocol's initialization, event dispatcher, and finalization functions. Therefore, to make these protocol functions available to the layer level functions, insert the following include statement in the file application.cpp:

```
#include "app_myprotocol.h"
```

4.2.5.3 Including MYPROTOCOL in List of Application Layer Protocols

Each node in EXata hosts an operating protocol stack. For each layer in the stack, a list of protocols running at that layer is maintained. When a new Application Layer protocol is added to EXata, it needs to be included in the list of Application Layer protocols. To do this, add the protocol name to the enumeration `AppType` defined in EXATA_HOME/include/application.h (see Section 4.2.3).

Traffic-generating applications have two parts: a client which generates the traffic and a server that receives the traffic. Both the client and server of an application protocol should be added to `AppType`.

For our example protocol, add the two entries `APP_MYPROTOCOL_CLIENT` (for the application client) and `APP_MYPROTOCOL_SEVER` (for the application server) to `AppType`, as shown in Figure 4-15.

```
typedef enum
{
    APP_FTP_SERVER_DATA = 20,
    APP_FTP_SERVER = 21,
    APP_FTP_CLIENT,
    APP_TELNET_SERVER = 23,
    APP_TELNET_CLIENT,
    ...
    /* Application-layer routing protocols */
    ...
    APP_ROUTING_FISHEYE = 160,      // IP protocol number
    APP_ROUTING_STATIC,
    ...
     APP_MYPROTOCOL_CLIENT,
     APP_MYPROTOCOL_SERVER,
    APP_PLACEHOLDER
} AppType;
```

FIGURE 4-15. Adding MYPROTOCOL to List of Application Layer Protocols

Note **Always add to the end of lists in header files (just before the entry `APP_PLACEHOLDER`).**

EXata provides for detailed traces of packets as they traverse the protocol stack at nodes in the network. A packet trace lists, among other information, the protocol that is handling the packet at the time of the trace. To facilitate tracing, EXata lists all protocols in an enumeration, `TraceProtocolType`, in the file EXATA_HOME/include/trace.h. For our example protocol, add an entry `TRACE_MYPROTOCOL` in `TraceProtocolType`, as shown in Figure 4-16.

```
typedef enum
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,                  // 1
    TRACE_UDP,                  // 2
    TRACE_IP,                   // 3
    TRACE_CBR,                  // 4
    TRACE_FTP,                  // 5
    ...
    TRACE_MYPROTCOL,
    // Must be last one!!!
    TRACE_ANY_PROTOCOL
}TraceProtocolType;
```

FIGURE 4-16. Adding MYPROTOCOL to List of Trace Protocols

Note **Always add to the end of lists in header files (just before the entry `TRACE_ANY_PROTOCOL`).**

4.2.5.4 Defining Data Structures

Each application has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Application parameters (see Section 4.2.5.5.2)
2. Application instance identification, such as port number (see Section 4.2.7.1)
3. Application state (see Section 4.2.5.5.3)
4. Statistics variables (see Section 4.2.5.7.1)

Define appropriate data structures for your application's client and server in the protocol header file, app_myprotocol.h. As an example, the following data structures (defined in app_cbr.h) are used by the CBR protocol:

1. `CbrData`: This is the main data structure used by the CBR protocol.

```
typedef struct struct_app_cbr_data
{
    short sourcePort;
    char type;
    Int32 seqNo;
    clocktype txTime;
    ...
} CbrData;
```

2. `AppDataCbrClient`: This data structure contains the CBR client information.

```
typedef struct struct_app_cbr_client_str
{
    Address localAddr;
    Address remoteAddr;
    D_Clocktype interval;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastSent;
    clocktype endTime;
    BOOL sessionIsClosed;
    D_Int64 numBytesSent;
    UInt32 numPktsSent;
    UInt32 itemsToSend;
    UInt32 itemSize;
    short sourcePort;
    Int32 seqNo;
    D_UInt32 tos;
}AppDataCbrClient;
```

3. `AppDataCbrServer`: This data structure contains the CBR server information.

```
typedef struct struct_app_cbr_server_str
{
    Address localAddr;
    Address remoteAddr;
    short sourcePort;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastReceived;
    BOOL sessionIsClosed;
    D_Int64 numBytesRecvd;
    UInt32 numPktsRecvd;
    clocktype totalEndToEndDelay;
    clocktype maxEndToEndDelay;
    clocktype minEndToEndDelay;
    Int32 seqNo;
    clocktype totalJitter;
    ...
} AppDataCbrServer;
```

4.2.5.5 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a traffic-generating Application Layer protocol.

4.2.5.5.1 Determining the Protocol Configuration Format

Each application has an input format for specifying user-specified configuration parameters. The application configuration is specified in the EXata application configuration file using this input format. The format for specifying an application's configuration parameters is:

```
<Protocol-name> <param1> <param2> ... <paramN>
```

where

`<Protocol-name>` : Unique identifier for the protocol.

`<param1>, ..., <paramN>` : User-specified configuration parameter values. An application protocol may have any number of required and/or optional parameters.

For example, to specify CBR traffic parameters in the configuration file EXATA_HOME/scenarios/default/default.app, use the following format:

```
CBR <src> <dest> <items_to_send> <item_size> <interval> <start time>
    <end time>
```

where

`<src>` : Client node's node identifier or IP address.

`<dest>` : Server node's node identifier or IP address.

`<items_to_send>` : Number of items to send.

`<item_size>` : Size of each item.

`<interval>` : Pause time between transmission of successive items.

`<start_time>` : Transmission start time.

`<end_time>` : Transmission end time.

The following example specifies that the node 1 will send 500 2-kilobyte items to node 2, sending one per minute, starting at 50 simulation seconds, and ending at 100 simulation seconds:

```
CBR 1 2 500 2048 1M 50S 100S
```

Decide on the format for specifying the new application's configuration parameters. For our example protocol, specify the configuration parameters in the EXata configuration file using the following format:

```
MYPROTOCOL <param1> <param2> ... <paramN>
```

Section 4.2.5.5.2 explains how to read user input specified in this format to initialize the application.

4.2.5.5.2 Reading Configuration Parameters and Calling the Protocol Initialization Function

EXata can configure a protocol to the parameters specified by the user in the EXata configuration file that sets up the experiment. This section explains how to read these user-specified configuration parameters for the application protocol and provide them to the protocol's initialization function.

The protocol stack of each node is initialized in a bottom up manner. The initialization of the Application Layer thus occurs after the other layers have been initialized. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the Application Layer initialization functions APP_InitializeApplications and APP_Initialize, which are implemented in the file application.cpp. APP_Initialize initializes Application Layer routing protocols while APP_InitializeApplications is used for initializing traffic-generating protocols.

Function APP_InitializeApplications reads the user's configuration parameters and passes them to the initialization functions of application protocols. To initialize a traffic-generating application protocol in EXata, add code to the function APP_InitializeApplications for reading the protocol's configuration parameters and for calling its initialization function.

Figure 4-17 shows how APP_InitializeApplications reads the configuration parameters for CBR and calls the initialization function for the CBR client and server. APP_InitializeApplications has access to the configuration input for the Application Layer specified by the user in the experiment configuration file. This input is stored in a variable called `appInput`. The first word of the currently examined input line is stored in a variable called `appStr`. The `appStr` variable is compared with the keyword used to uniquely identify an application, such as CBR, FTP, etc. If a match occurs, then the parameters of the application are read from the input string. The C library function `sscanf` is used to split the input string into multiple words that constitute the parameters of the protocol.

The source and destination node identifiers are parameters commonly specified by users for most traffic-generating applications. The strings containing the source and destination information (obtained by splitting the input string) are passed to the EXata library function IO_AppParseSourceAndDestStrings, defined in ð/include/fileio.h. This function performs the following tasks:

1. Gets the source and destination node identifier and node address from the input strings
2. Displays an error message if the source or destination does not exist

After obtaining the node identifier, the EXata library function MAPPING_GetNodePtrFromHash, defined in EXATA_HOME/include/mapping.h, is called to get a handle to the node pointer which stores the state of the source node.

The parameters containing time related information are converted from string to EXata's clocktype variables by calling EXata library function TIME_ConvertToClock defined in EXATA_HOME/include/clock.h.

This is followed by a call to the CBR client initialization function AppCbrClientInit, which is passed the source node pointer and user configuration values that were read from input string `appInput`. Then, APP_InitializeApplications calls function APP_SuccessfullyHandledLoopback to check if the application is specified as a loopback application. If it is not a loopback application, function MAPPING_GetNodePtrFromHash is called to get the destination node pointer. Lastly, the CBR server initialization function, AppCbrServerInit, is called with destination node pointer as the parameter. The initialization functions AppCbrClientInit and AppCbrServerInit are implemented in the file app_cbr.cpp. Function APP_SuccessfullyHandledLoopback is implemented in application.cpp.

```
void APP_InitializeApplications(Node *firstNode, const NodeInput *nodeInput)
{
   NodeInput appInput;
   char appStr[MAX_STRING_LENGTH];
   ...
   for (i = 0; i < appInput.numLines; i++)
    {
        sscanf(appInput.inputStrings[i], "%s", appStr);
        ...
        else
        if (strcmp(appStr, "CBR") == 0)
        {
            char sourceString[MAX_STRING_LENGTH];
            ...
            NodeAddress sourceNodeId;
            Address sourceAddr;
            ...
            numValues = sscanf(appInput.inputStrings[i],
                               "%*s %s %s %d %d %s %s %s %s %s %s",
                               sourceString, destString, &itemsToSend,
                               &itemSize, intervalStr, startTimeStr,
                               endTimeStr, optionToken1, optionToken2,
                               optionToken3);
            ...
            IO_AppParseSourceAndDestStrings(
                firstNode, appInput.inputStrings[i], sourceString,
               &sourceNodeId, &sourceAddr, destString, &destNodeId, &destAddr);
            node = MAPPING_GetNodePtrFromHash(nodeHash, sourceNodeId);
            if (node != NULL)
            {
                clocktype startTime = TIME_ConvertToClock(startTimeStr);
                clocktype endTime = TIME_ConvertToClock(endTimeStr);
                ...
                AppCbrClientInit(node, sourceAddr, destAddr, itemsToSend
                                 itemSize, interval, startTime, endTime,
                                 tos, isRsvpTeEnabled);
            }
            ...
            // Handle Loopback Address
            if (node == NULL ||
                !APP_SuccessfullyHandledLoopback(
                                    node, appInput.inputStrings[i], destAddr,
                                    destNodeId, sourceAddr, sourceNodeId))
            {
                node = MAPPING_GetNodePtrFromHash(nodeHash, destNodeId);
            }
            if (node != NULL)
            {
                AppCbrServerInit(node);
            }
        }
        ...
}
```

FIGURE 4-17. Calling CBR Initialization Functions

Add code to the function APP_InitializeApplications to read the configuration parameters for the application MYPROTOCOL from the input file, and to call the initialization functions for the client and server, AppMyprotocolClientInit and AppMyprotocolServerInit, respectively. Figure 4-18 shows an outline of the code that should be added.

```
void APP_InitializeApplications(Node *firstnode, const NodeInput *nodeInput)
{
   NodeInput appInput;
   char appStr[MAX_STRING_LENGTH];
   ...
   for (i = 0; i < appInput.numLines; i++)
    {
        sscanf(appInput.inputStrings[i], "%s", appStr);
        ...
        else
        if (strcmp(appStr, "CBR") == 0)
        {
          ...
        }
        else
          if (strmcp(appStr, "MYPROTOCOL") == 0)
          {
              /* Initialize variables for reading user input */
              ...
              /*Read user input into appropriate variables */
              retVal == sccanf(appInput.inputStrings[i], ...);
              ...
              /* Get source and destination nodeId and address */
              IO_AppParseSourceAndDestStrings(...)
              ...
              /* Get the pointer to the source node */
              node = MAPPING_GetNodePtrFromHash (...);
              if (node != NULL)
              {
                  ...
                  /* Call MYPROTOCOL client initialization function */
                  AppMyprotocolClientInit (node, ...);
              }
              /* Get the pointer to the destination node */
              node = MAPPING_GetNodePtrFromHash (...);
              if (node != NULL)
              {
                  ...
                  /* Call MYPROTOCOL server initialization function */
                  AppMyprotocolServerInit (node, ...);
              }
          }
        ...
    }
}
```

FIGURE 4-18. Calling MYPROTOCOL Initialization Functions

4.2.5.5.3 Implementing the Client Initialization Function

The initialization of an application takes place in the initialization function of the protocol that is called by the Application Layer initialization function APP_InitializeApplications. The initialization function of an application commonly performs the following tasks:

- Initialize the state and store the user specified configuration parameters
- Initialize data structures and variables as required, e.g., allocate memory to tables, set default values, etc.
- Create an instance of the application
- Schedule a timer to itself for starting the application, if the application uses UDP at the Transport Layer
- Open a TCP connection, if the application uses TCP at the Transport Layer

This section describes how to initialize the client for a UDP-based application. For an example of initializing the client for a TCP-based application, refer to the FTP function AppFtpClientInit in EXATA_HOME/libraries/developer/src/app_ftp.cpp.

Like all other functions belonging to the application, the prototype for the initialization functions should be included in the application's header file, app_myprotocol.h.

4.2.5.5.3.1 Creating an Instance and Initializing the State

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as timer state (next periodic update, etc.), flags, connection information, sequence number, pointers to tables used by the protocol, etc. Each instance of the application maintains its own state variable.

To store the state, declare the structure to hold the protocol state in the header file, app_myprotocol.h (see Section 4.2.5.4).

Create an instance of the application by allocating memory to the state structure. CBR performs this task by calling the function AppCbrClientNewCbrClient in its initialization function AppCbrClientInit, as shown in Figure 4-19. `AppDataCbrClient` is the data structure for the CBR client (see Section 4.2.5.4).

```
void AppCbrClientInit(Node *node, Address clientAddr, Address serverAddr,
                      Int32 itemsToSend, Int32 itemSize, clocktype interval,
                      clocktype startTime, clocktype endTime, unsigned tos,
                      BOOL isRsvpTeEnabled)
{
    ...
    AppDataCbrClient *clientPtr; //pointer to the state structure
    ...
    clientPtr = AppCbrClientNewCbrClient(
                            node,
                            clientAddr, serverAddr,
                            itemsToSend, itemSize,
                            interval, startTime,
                            endTime, (TosType) tos);
    ...
}
```

FIGURE 4-19. Creating an Application Instance in Initialization Function

Function AppCbrClientNewCbrClient calls the function MEM_malloc to allocate memory to the state structure. It then stores the user specified configuration parameters that were passed to the initialization function AppCbrClientInit as shown in Figure 4-20.

```
AppDataCbrClient *AppCbrClientNewCbrClient(
                     Node *node, Address localAddr, Address remoteAddr,
                     Int32 itemsToSend, Int32 itemSize, clocktype interval,
                     clocktype startTime, clocktype endTime, TosType tos)

{
    AppDataCbrClient *cbrClient;

    cbrClient = (AppDataCbrClient*)
                MEM_malloc(sizeof(AppDataCbrClient));
   memset(cbrClient, 0, sizeof(AppDataCbrClient));
    /*
      * fill in cbr info.
      */
    ...
    cbrClient->interval = interval;
    cbrClient->sessionStart = getSimTime(node) + startTime;
    ...
    cbrClient->sourcePort = node->appData.nextPortNum++;
    ...
    APP_RegisterNewApp(node, APP_CBR_CLIENT, cbrClient);
    return cbrClient;
}
```

FIGURE 4-20. Function to Create and Initialize an Application Instance

Multiple instances of the same application may run at a node. Therefore, the protocol state structure must have an identifying field. For the CBR client application, this is the field `sourcePort` of the `AppDataCbrClient` data structure. When a new instance of the CBR client is created, this field is assigned the next available port number, as shown in Figure 4-20. This ensures that a unique port number is associated with each instance.

4.2.5.5.3.2 Registering the Application

The next step after creating the application instance is to register the instance as one of the protocols running at the node. This is done by making a call to APP_RegisterNewApp, as shown in Figure 4-20. Function APP_RegisterNewApp is a EXata library function (defined in app_util.cpp) to add an application to the list of applications running at the node. When the application needs to access its state variable, it retrieves the state variable from this list. Each element of this list is of the type `AppInfo` (see Section 4.2.3).

APP_RegisterNewApp accepts the following parameters:

- the node pointer
- the application type
- the pointer to the state structure

4.2.5.5.3.3 Initializing Timers

Besides initializing data structures, the initialization function also initializes timers for the application. Timers serve a variety of purposes at the Application Layer, e.g., to notify when the application is supposed to begin sending data, to simulate traffic sending rate, etc.

This section discusses in detail how to use timers. Since each node can have multiple applications of the same type, Application Layer timers frequently use the message `info` field to identify which application instance the timer is for.

EXata provides a general structure used to hold information on application timers called `AppTimer` (see Section 4.2.3). `AppTimer` can be used to store the following information:

- Timer Type: Category or purpose of timer
- Connection Id: Connection this timer is meant for
- SourcePort: The session that this timer belongs to.

The timer type can be one of the following three pre-defined types:

- Name: `APP_TIMER_SEND_PKT`
  Purpose: Timer to send a packet. Used to simulate data sending rate.
- Name: `APP_TIMER_UPDATE_TABLE`
  Purpose: Timer to update a local table, e.g., update entries, remove timed-out entries from a table, etc.
- Name: `APP_TIMER_CLOSE_SESS`
  Purpose: Timer to close a session.

Figure 4-21 shows the code from the initialization function, AppCbrClientInit, that sets a timer to inform the CBR client of when to start sending data. It demonstrates how a timer can store the source port of the application instance in the message `info` field. This source port is used to identify the instance of the CBR application, in case there are multiple CBR applications running at the node. The timer type used is `APP_TIMER_SEND_PKT` because the purpose of the timer is to tell the CBR client to send a packet.

```
void AppCbrClientInit(Node *node, Address clientAddr, Address serverAddr,
                      Int32 itemsToSend, Int32 itemSize, clocktype interval,
                      clocktype startTime, clocktype endTime, unsigned tos,
                      BOOL isRsvpTeEnabled)
{
    ...
    AppTimer *timer;
    ...
    Message *timerMsg;
    ...
    timerMsg = MESSAGE_Alloc(node,
                             APP_LAYER,
                             APP_CBR_CLIENT,
                             MSG_APP_TimerExpired);
    MESSAGE_InfoAlloc(node, timerMsg, sizeof(AppTimer));
    timer = (AppTimer *)MESSAGE_ReturnInfo(timerMsg);
    timer->sourcePort = clientPtr->sourcePort;
    timer->type = APP_TIMER_SEND_PKT;
    MESSAGE_Send(node, timerMsg, startTime);
}
```

FIGURE 4-21. Initializing Timers

The message type used here is `MSG_APP_TimerExpired`. Commonly needed message types for the Application Layer are defined in api.h.

The API function APP_SetTimer can also be used instead of the code in Figure 4-21 to set a new Application Layer timer and send to self after a specified delay. APP_SetTimer is implemented in app_util.cpp.

4.2.5.5.4 Implementing the Server Initialization Function

For a UDP-based application, such as CBR, the server is initialized when it receives the first packet from the client. This is discussed in Section 4.2.5.6.3.

For an example of initializing the server for a TCP-based application, refer to the FTP function AppFtpServerInit in app_ftp.cpp.

4.2.5.6 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a traffic-generating protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the Application Layer, NODE_ProcessEvent calls the Application Layer event dispatcher APP_ProcessEvent, defined in application.cpp.

Section 4.2.5.6.1 describes how to modify the Application Layer event dispatcher function to call the traffic-generating protocol's event dispatchers. Section 4.2.5.6.2 and Section 4.2.5.6.3 describe how to implement the event dispatcher for the protocol client and the protocol server, respectively.

4.2.5.6.1 Modifying the Application Layer Event Dispatcher

Function APP_ProcessEvent implements the Application Layer event dispatcher that informs the appropriate application protocol of received events. Messages contain the name of the protocol they are destined for. (This is the application protocol name specified in the enumerated data type `AppType`, described in Section 4.2.3.) The API function APP_GetProtocolType returns the protocol for which the message is destined. APP_ProcessEvent implements a switch statement on the protocol name read from the message and calls the appropriate protocol-specific event dispatcher.

To enable the protocol MYPROTOCOL to receive events, add code to APP_ProcessEvent to call the protocol's event dispatcher function when messages for the protocol are received. For a traffic-generating application protocol, do this separately for both the server and the client. Figure 4-22 shows a code fragment from APP_ProcessEvent with sample code for calling the client event dispatcher function, AppLayerMyprotocolClient, and the server event dispatcher function, AppLayerMyprotocolServer.

```
void APP_ProcessEvent(Node *node, Message *msg)
{
    short protocolType;
    protocolType = APP_GetProtocolType(node,msg);
    switch(protocolType)
    {
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
        ...
        case APP_CBR_CLIENT:
        {
            AppLayerCbrClient(node, msg);
            break;
        }
        case APP_CBR_SERVER:
        {
            AppLayerCbrServer(node, msg);
            break;
        }
        case APP_MYPROTOCOL_CLIENT:
        {
            AppLayerMyprotocolClient(node, msg);
            break;
        }
        case APP_MYPROTOCOL_SERVER:
        {
            AppLayerMyprotocolServer(node, msg);
            break;
        }
        ...
 }//switch//
}
```

FIGURE 4-22. Calling MYPROTOCOL Event Dispatcher Functions

4.2.5.6.2 Implementing the Client Event Dispatcher

A protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received.

All event types used by EXata protocols are enumerated in the file api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as shown in Figure 4-23.

```
// /**
// ENUM        :: MESSAGE/EVENT
// DESCRIPTION :: Event/message types exchanged in the simulation
// **/
enum
{
    /* Special message types used for internal design. */
    MSG_SPECIAL_Timer                          = 0,
    ...
    /* Message Types for Channel layer */
    MSG_PROP_SignalArrival                     = 100,
    MSG_PROP_SignalEnd                         = 101,
    ...
    /*
     * Any other message types which have to be added should be added before
     * MSG_DEFAULT. Otherwise the program will not work correctly.
     */
    MSG_APP_MYPROTOCOL_NewEvent1,
    MSG_APP_MYPROTOCOL_NewEvent2,
    MSG_DEFAULT                                = 10000
};
```

FIGURE 4-23. Declaring New Event Types

Note **Always add to the end of lists in header files (just before the entry `MSG_DEFAULT`).**

To understand how a protocol event dispatcher works, we examine the code for the function AppLayerCbrClient, which is the event dispatcher for the CBR client. Function AppLayerCbrClient and the other functions used by the CBR application are implemented in the file app_cbr.cpp.

When an event occurs, the first thing to do is to determine which instance of the application protocol this event is for. This can be done by looking up additional information stored in the message `info` field, such as source or destination port. For example, the CBR function AppCbrClientGetCbrClient searches the list of application instances running at the node, based on the source port number, and returns the data structure for the appropriate application instance, as shown in Figure 4-24.

```
AppDataCbrClient *
AppCbrClientGetCbrClient(Node *node, short sourcePort)
{
    AppInfo *appList = node->appData.appPtr;
    AppDataCbrClient *cbrClient;

    for (; appList != NULL; appList = appList->appNext)
    {
        if (appList->appType == APP_CBR_CLIENT)
        {
            cbrClient = (AppDataCbrClient *) appList->appDetail;
            if (cbrClient->sourcePort == sourcePort)
            {
                return cbrClient;
            }
        }
    }

    return NULL;
}
```

FIGURE 4-24. Searching the List of Application Instances

CBR operates by setting a periodic timer to itself. Each time the timer goes off, the client sends a data packet to the destination. It then sets a new timer to occur after the periodic interval. In this way the desired data rate is achieved. The timer is initialized in the initialization function for the CBR Client, AppCbrClientInit, where a timer of type `APP_TIMER_SEND_PKT` is set for the start time of the CBR application (see Section 4.2.5.5.3.3). Figure 4-25 shows the code to handle this timer event in the CBR client event dispatcher function AppLayerCbrClient.

The API function APP_UdpSendNewHeaderVirtualDataWithPriority sends a packet to UDP at the Transport Layer. (See the *API Reference Guide* for the complete list of Application Layer APIs and the file app_util.cpp for their implementation.) UDP delivers the packet to the application protocol (CBR server, in this case) at the destination node.

Instead of the layer-specific APIs, such as APP_UdpSendNewHeaderVirtualDataWithPriority, message APIs can be used to communicate between layers, as discussed in Section 3.3.1.2.

After sending a packet, the CBR client determines if any more packets need to be sent. If this is the case, it calls the function AppCbrClientScheduleNextPkt. Function AppCbrClientScheduleNextPkt sets a timer of type `APP_TIMER_SEND_PKT` to occur after the inter-packet interval of the CBR application.

After the message is handled by the event dispatcher, it frees the memory associated with the message by calling the function MESSAGE_Free.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

The event dispatcher also includes a default case in the switch statement to handle messages which contain an undefined event type.

```
void AppLayerCbrClient(Node *node, Message *msg)
{
    ...
    AppDataCbrClient *clientPtr;
    ...
    switch(msg->eventType)
    {
        case MSG_APP_TimerExpired:
        {
            AppTimer *timer;
            timer = (AppTimer *) MESSAGE_ReturnInfo(msg);
            ...
            clientPtr = AppCbrClientGetCbrClient(node, timer->sourcePort);
            ...
            switch (timer->type)
            {
                case APP_TIMER_SEND_PKT:
                {
                    CbrData data;
                    ...
                    data.sourcePort = clientPtr->sourcePort;
                    data.txTime = getSimTime(node);
                    data.seqNo = clientPtr->seqNo++;
                    ...
                    APP_UdpSendNewHeaderVirtualDataWithPriority(
                        node,
                        APP_CBR_SERVER,
                        clientPtr->localAddr,
                        (short) clientPtr->sourcePort,
                        clientPtr->remoteAddr,
                        (char *) &data,
                        sizeof(data),
                        clientPtr->itemSize - sizeof(data),
                        clientPtr->tos,
                        0,
                        TRACE_CBR);
                    ...
                  }
               default:
                    assert(FALSE);
            }
            break;
        }
        default:
           // Print error message
           ...
           ERROR_ReportError(error);
    }
    MESSAGE_Free(node, msg);
}
```

FIGURE 4-25. Event Dispatcher for CBR Client

Use of virtualPayLoadSize Field

For some applications, the contents of part or all of the user data are not important in the simulation of the application and need not be explicitly stored in the packet field of a message. However, the size of the user data is important because it affects the calculation of transmission time and required buffer space at lower layers. The virtualPayLoadSize field of the message data structure is used to store the size of the part of user data whose contents are not important. Therefore, using the virtualPayLoadSize field saves memory.

In CBR simulation, the contents of the user data are not important. However, an application header is used in the simulation to store the source port number, the sequence number of data item being transmitted, and the time when the data item is transmitted. This header information does not correspond to actual CBR data and is meant for simulator use only. The header is stored in the packet field of a message. The difference between the CBR item size being simulated and the size of the application header is stored in the virtualPayLoadSize field of the message. This is done in function APP_UdpSendNewHeaderVirtualDataWithPriority by calling the function MESSAGE_AddVirtualPayload.

4.2.5.6.3 Implementing the Server Event Dispatcher

We use the CBR application as an example to understand the server event dispatcher, as we did for the client dispatcher. Function AppLayerCbrServer is the event dispatcher for the CBR server. This function is implemented in the file app_cbr.cpp, and snippets from it are shown in Figure 4-26.

The CBR server receives packets from the client and processes them. When a packet arrives at the destination node, it travels up the protocol stack one layer at a time. The UDP protocol at the Transport Layer sends the packet to the CBR server at the Application Layer by scheduling an event of the type MSG_APP_FromTransport at the Application Layer.

The CBR server event dispatcher performs actions corresponding to the event type of the received message. Event MSG_APP_FromTransport indicates the arrival of a packet from the Transport Layer. The CBR server handles this event by processing the received packet.

When a packet arrives at the CBR server, function AppLayerCbrServer first determines the protocol instance for which the packet is destined. AppLayerCbrServer calls function AppCbrServerGetCbrServer to search the list of application instances running at the node, based on the source address and source port number. If AppCbrServerGetCbrServer finds a match, it returns a pointer to the data structure for the appropriate instance; otherwise, it returns NULL. If AppCbrServerGetCbrServer returns NULL, it indicates that the received packet is for a new connection and AppLayerCbrServer initiates a new instance for the CBR server by calling function AppCbrServerNewCbrServer. Functions AppCbrServerGetCbrServer and AppCbrServerNewCbrServer are similar to the corresponding functions for the CBR client, AppCbrClientGetCbrClient (see Section 4.2.5.6.2) and AppCbrClientNewCbrClient (see Section 4.2.5.5.3.1) and are implemented in app_cbr.cpp.

Function MESSAGE_ReturnPacket returns the packet field of a message and function MESSAGE_ReturnPacketSize returns the size of the packet field.

As in the case of the client event dispatcher, after the message is handled by the server event dispatcher, the server dispatcher frees the memory associated with the message by calling the function MESSAGE_Free.

The event dispatcher also includes a default case in the switch statement to handle messages which contain an undefined event type.

```
void AppLayerCbrServer(Node *node, Message *msg
{
    char error[MAX_STRING_LENGTH];
    AppDataCbrServer *serverPtr;

    switch(msg->eventType)
    {
        case MSG_APP_FromTransport:
        {
            UdpToAppRecv *info;
            CbrData data;

            info = (UdpToAppRecv *) MESSAGE_ReturnInfo(msg);
            memcpy(&data, MESSAGE_ReturnPacket(msg), sizeof(data));
            ...
            serverPtr = AppCbrServerGetCbrServer(node,
                                                 info->sourceAddr,
                                                 data.sourcePort);

            /* New connection, so create new CBR server to handle client. */
            if (serverPtr == NULL)
            {
                serverPtr = AppCbrServerNewCbrServer(node,
                                                     info->destAddr,
                                                     info->sourceAddr,
                                                     data.sourcePort);
            }
            ...
            if (data.seqNo >= serverPtr->seqNo)
            {
                serverPtr->numBytesRecvd += MESSAGE_ReturnPacketSize(msg);
                serverPtr->sessionLastReceived = getSimTime(node);
                ...
                serverPtr->seqNo = data.seqNo + 1;
                ...
             }
             ...
            break;
        }
        default:
        {
            ...
            ERROR_ReportError(error);
        }
    }

    MESSAGE_Free(node, msg);
}
```

FIGURE 4-26. Event Dispatcher for CBR Server

All event types used by EXata protocols are enumerated in the file api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as described in Section 4.2.5.6.2.

4.2.5.7 Collecting and Reporting Statistics

In this section, we describe how to collect and report statistics for a traffic-generating Application Layer protocol.

4.2.5.7.1 Declaring Statistics Variables

An application protocol can be configured to record statistics specified by the programmer, such as:

- Number of bytes sent
- Number of bytes received
- Number of packets sent

To enable statistics collection for the protocol, include the statistic collection variables in the structure used to hold the protocol state (see Section 4.2.5.4). For example, the data structure for the CBR server, `AppDataCbrServer`, defined in app_cbr.h, includes statistics variables such as:

- `numBytesRecvd`: Variable to record number of received bytes
- `numPktsRecvd`: Variable to record number of received packets
- `totalEndToEndDelay`: Variable used to calculate throughput

The statistics related variables can also be defined in a structure and then that structure is included in the state variable.

4.2.5.7.2 Initializing Statistics

Initialize statistics variables in the function that initializes an instance of the protocol. For example, function AppCbrServerNewCbrServer in file app_cbr.cpp is the function that creates and initializes an instance of the CBR server. It also initializes the statistics variables that are declared as part of the CBR server data structure, `AppDataCbrServer`. A code snippet from function AppCbrServerNewCbrServer is in Figure 4-27.

```
AppDataCbrServer *
AppCbrServerNewCbrServer(Node *node, Address localAddr,
                         Address remoteAddr, short sourcePort)
{
   AppDataCbrServer *cbrServer;
   cbrServer = (AppDataCbrServer *)
               Mem_alloc(sizeof(AppDataCbrServer));
   ...
   cbrServer->numBytesRecvd = 0;
   cbrServer->numPktsRecvd = 0;
   cbrServer->totalEndToEndDelay = 0;
   ...
}
```

FIGURE 4-27. Initializing Statistics Variables

4.2.5.7.3 Updating Statistics

After declaring and initializing the statistics variables, update their value during the protocol life cycle, as required. For example, increment the value of `numBytetsRecvd` every time the receiver gets a packet. The CBR server function AppLayerCbrServer (see Figure 4-26) performs this task by executing the following code when the server receives a packet:

```
serverPtr->numBytesRecvd += MESSAGE_ReturnPacketSize(msg);
serverPtr->sessionLastReceived = getSimTime(node);
```

The API function MESSAGE_ReturnPacketSize returns the size of the packet associated with a message.

4.2.5.7.4 Printing Statistics

As a final step towards statistics collection, create a function to print the client statistics and a function to print the server statistics. These print functions are called in the finalization functions of the protocol, which are discussed in Section 4.2.5.7.5.

Function AppCbrServerPrintStats, shown in Figure 4-28, calls the C function sprintf to create a single string containing the statistic name and statistic value, and then calls function IO_PrintStat to print that string to a file. Function IO_PrintStat function, defined in EXATA_HOME/include/fileio.h, requires the following parameters:

- Node pointer: Pointer to the node reporting the statistics.
- Layer: String indicating the layer. Set this to `"Application"` for the Application Layer.
- Protocol: String indicating the protocol name.
- Interface address: Interface address. Set this to `ANY_DEST` for Application Layer protocols.
- Instance identifier: Instance identifier or port number.
- Buffer: String containing the statistics.

```
void AppCbrServerPrintStats(Node *node, AppDataCbrServer *serverPtr)
{
    clocktype throughput;
    ...
    char buf[MAX_STRING_LENGTH];
    ...
    sprintf(buf, "Total Packets Received = %u", serverPtr->numPktsRecvd);
    IO_PrintStat(
        node,
        "Application",
        "CBR Server",
        ANY_DEST,
        serverPtr->sourcePort,
        buf);

    sprintf(buf, "Throughput (bits/s) = %s", throughputStr);
    IO_PrintStat(
        node,
        "Application",
        "CBR Server",
        ANY_DEST,
        serverPtr->sourcePort,
        buf);
   ...
}
```

FIGURE 4-28. Function to Print Statistics

4.2.5.7.5 Adding Dynamic Statistics

Dynamic statistics are statistic variables whose values can be observed in the EXata GUI during the simulation. See Section 5.2.3 for adding dynamic statistics to a protocol. Refer to *EXata User's Guide* for details of viewing dynamic statistics during the simulation.

4.2.5.8 Finalization

The finalization function of the protocol is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

At the end of simulation, the finalization function for each protocol is called to print the protocol statistics. As discussed in Section 3.4.3, the finalization function is called hierarchically. The node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp, calls the finalization function for Application Layer, APP_Finalize, defined in application.cpp. APP_Finalize calls the finalization function(s) of each application protocol running at the node.

4.2.5.8.1 Modifying the Application Layer Finalization Function

Call the finalization function(s) of the application protocol from the Application Layer finalization function, APP_Finalize, defined in application.cpp. To add the protocol's finalization function, add a case statement on the protocol name and make a call to the finalization function within the case, as done for other protocols in the function. In APP_Finalize, the finalization functions of Application Layer routing models are specified in the first switch statement and the second switch statement is used to specify finalization functions of Application Layer traffic generators. Figure 4-29 shows the outline of code that needs to be added to APP_Finalize. Function AppMyprotocolClientFinalize is the finalization function for MYPROTOCOL client (see Section 4.2.5.8.2) and function AppMyprotocolServerFinalize is the finalization function for MYPROTOCOL server (see Section 4.2.5.8.3).

```
void APP_Finalize (Node *node)
{
  ...
  AppInfo *applist = NULL;
  AppInfo *nextApp = NULL;
  ...
  for (appList = node->appData.appPtr; appList != NULL;
       appList = nextApp)
    {
        switch (appList->appType)
        {
         ...
         case APP_CBR_CLIENT:
            {
                AppCbrClientFinalize(node, appList);
                break;
            }
          case APP_CBR_SERVER:
            {
                AppCbrServerFinalize(node, appList);
                break;
            }
          case APP_MYPROTOCOL_CLIENT
            {
                AppMyprotocolClientFinalize(node, appList);
                break;
            }
          case APP_MYPROTOCOL_SERVER:
            {
                AppMyprotocolServerFinalize(node, appList);
                break;
            }
         ...
         nextApp = appList->appNext;
    }
  ...
}
```

FIGURE 4-29. Calling MYPROTOCOL Finalization Functions

4.2.5.8.2 Implementing the Client Finalization Function

Write the finalization function for protocol client, AppMyprotocolClientFinalize. If statistics collection is enabled for the Application Layer, call the function to print the client's statistics (see Section 4.2.5.7.4) from the finalization function, or add code directly to AppMyprotocolClientFinalize to print statistics.

Use the CBR client finalization function, AppCbrClientFinalize, shown in Figure 4-30, as a template. This function is implemented in app_cbr.cpp.

```
void AppCbrClientFinalize(Node *node, AppInfo* appInfo)
{
    AppDataCbrClient *clientPtr =
                    (AppDataCbrClient*)appInfo->appDetail;

    if (node->appData.appStats == TRUE)
    {
        AppCbrClientPrintStats(node, clientPtr);
    }
}
```

FIGURE 4-30. Finalization Function for CBR Client

As for all other functions, specify the prototype of the finalization function in the protocol's header file, app_myprotocol.h.

4.2.5.8.3 Implementing the Server Finalization Function

Write the finalization function for protocol client, AppMyprotocolServerFinalize. If statistics collection is enabled for the Application Layer, call the function to print the client's statistics (see Section 4.2.5.7.4) from the finalization function, or add code directly to AppMyprotocolServerFinalize to print statistics.

Use the CBR client finalization function, AppCbrServerFinalize, shown in Figure 4-31, as a template. This function is implemented in app_cbr.cpp.

```
void AppCbrServerFinalize(Node *node, AppInfo* appInfo)
{
  AppDataCbrServer *serverPtr = (AppDataCbrServer*)appInfo->appDetail;
  if (node->appData.appStats == TRUE)
  {
     AppCbrServerPrintStats(node, serverPtr);
  }
}
```

FIGURE 4-31. Finalization Function for CBR Server

As for all other functions, specify the prototype of the finalization function in the protocol's header file, app_myprotocol.h.

4.2.5.9 Including and Compiling Files

The final step in integrating your application model into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the application model in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Developer library, then modify EXATA_HOME/libraries/developer/Makefile-common as shown in Figure 4-32. Recompile EXata after making the changes.

```
...
# common sources
#
DEVELOPER_SRCS = \
$(DEVELOPER_SRCDIR)/adaptation_aal5.cpp \
$(DEVELOPER_SRCDIR)/adaptation.cpp \
...
$(DEVELOPER_SRCDIR)/app_mcbr.cpp \
$(DEVELOPER_SRCDIR)/app_messenger.cpp \
$(DEVELOPER_SRCDIR)/app_myprotocol.cpp \
$(DEVELOPER_SRCDIR)/app_superapplication.cpp \
$(DEVELOPER_SRCDIR)/app_telnet.cpp \
$(DEVELOPER_SRCDIR)/app_traffic_gen.cpp \
...
```

FIGURE 4-32. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.2.5.10 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.2.6 Adding an Application Layer Routing Protocol

Application Layer routing protocols function mostly as other Application Layer protocols. They operate at the Application Layer and send/receive packets using Application Layer APIs. However, there are certain differences in their implementation. This section requires knowledge of the contents of Section 4.2.5 and provides additional implementation details needed for implementing a routing protocol at the Application Layer.

To understand how an Application Layer routing protocol is implemented in EXata, look at the implementation code for the routing protocol Bellman-Ford. The header file for the Bellman-Ford implementation is routing_bellmanford.h and the source file is routing_bellmanford.cpp in the folder EXATA_HOME/libraries/developer/src.

The following list summarizes the actions that need to be performed for adding an Application Layer routing protocol, MYPROTCOL, to EXata. For those steps that are similar to the steps for writing a traffic-generating protocol, we refer the reader to the appropriate subsection of Section 4.2.5. The steps that are different for routing protocols are described in detail in subsequent sections. (Note that unlike a traffic-generating Application Layer protocol, which has a client and a server, a routing protocol has a single module.)

1. Create header and source files (see Section 4.2.5.2).
2. Modify the file application.cpp to include the protocol's header file (see Section 4.2.5.2).
3. Include the protocol in the list of Application Layer protocols and trace protocols (see Section 4.2.6.1).
4. Modify Application Layer data structure to include routing protocol state (see Section 4.2.6.2)
5. Include the protocol in Network Layer declarations (see Section 4.2.6.3).
6. Define data structures for the protocol (see Section 4.2.5.4).
7. Decide on the format for the protocol-specific configuration parameters (see Section 4.2.6.4.1).
8. Call the protocol's initialization function from the Application Layer initialization function, APP_Initialize (see Section 4.2.6.4.2).
9. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Read and store the configuration parameters (see Section 4.2.6.4.3.1).
   b. Initialize the state variables (see Section 4.2.6.4.3.1).
   c. Initialize timers (see Section 4.2.6.4.3.2).
   d. Initialize routing tables (see Section 4.2.6.4.3.3).
10. Integrate the protocol with the Network Layer (see Section 4.2.6.5).
    a. Modify the IP function NetworkRoutingGetAdmindistance to return the protocol's administrative distance.
    b. Modify the routing protocol parsing function NetworkIpParseAndSetRoutingProtocolType to include the protocol in the list of routing protocols that are initialized at the Application Layer.
11. Call the protocol event dispatcher from the Application Layer event dispatcher, APP_ProcessEvent (see Section 4.2.6.6.1).
12. Declare any new event types used by the protocol in the header file EXATA_HOME/include/api.h (see Section 4.2.5.6.2).
13. Write the protocol event dispatcher (see Section 4.2.6.6.2).
14. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.2.5.7.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.2.5.7.2.)
    c. Update the statistics as appropriate (see Section 4.2.5.7.3.)
    d. Write a function to print the statistics (see Section 4.2.5.7.4.)
    e. Add dynamic statistics to the protocol, if desired (see Section 4.2.5.7.5) .
15. Call the protocol finalization function from the Application Layer finalization function, APP_Finalize (see Section 4.2.6.8.1).
16. Write the protocol finalization function (see Section 4.2.6.8.2). Call the function to print statistics from the protocol finalization function.
17. Include the protocol header and source files in the EXata tree and compile (see Section 4.2.5.9).
18. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.2.5.10).

4.2.6.1 Including MYPROTOCOL in List of Application Layer Protocols

This step is similar to the corresponding step for adding a traffic-generating protocol (see Section 4.2.5.3), except that only one entry needs to be made to the list of protocols.

For our example protocol MYPROTOCOL, add `APP_MYPROTOCOL` to `AppType`, as shown in Figure 4-33. Enumeration `AppType` is defined in application.h.

```
typedef enum
{
    APP_FTP_SERVER_DATA = 20,
    APP_FTP_SERVER = 21,
    APP_FTP_CLIENT,
    APP_TELNET_SERVER = 23,
    ...
    APP_MYPROTOCOL,
    APP_PLACEHOLDER
} AppType;
```

FIGURE 4-33. Adding MYPROTOCOL to List of Application Layer Routing Protocols

Note **Always add to the end of lists in header files (just before the entry `APP_PLACEHOLDER`).**

As in the case of a traffic-generating protocol, add an entry `TRACE_MYPROTOCOL` in the enumeration `TraceProtocolType` in EXATA_HOME/include/trace.h, as shown in Figure 4-16.

4.2.6.2 Modify AppData to include MYPROTOCOL State Information

The routing protocol state is stored in the Application Layer data structure `AppData`. To add a custom Application Layer routing protocol to EXata, modify `AppData` (Section 4.2.3) to include the protocol's state, as shown in Figure 4-34.

```
struct struct_app_str
{
    AppInfo *appPtr;      /* pointer to the list of app info */
    PortInfo *portTable;  /* pointer to the port table */
    short  nextPortNum;   /* next available port number */
    BOOL   appStats;      /* whether application statistics
                             collection is enabled */
    AppType exteriorGatewayProtocol;
    BOOL routingStats;
    void *routingVar;
    void *bellmanford;
    void *olsr;
    void *olsr2;
    void *myprotocol;
   ...
  };
```

FIGURE 4-34. Modifying AppData to include Routing Protocol State

4.2.6.3 Including MYPROTOCOL in Network Layer Declarations

Each node in EXata maintains a list of routing protocols running at the node. When a new Application Layer routing protocol is added to EXata, it needs to be included in the list of routing protocols. To do this, add the protocol name to the enumeration `NetworkRoutingProtocolType` defined in EXATA_HOME/include/network.h (see Section 4.4.3).

For our example protocol, add the entry `ROUTING_PROTOCOL_MYPROTOCOL` to `NetworkRoutingProtocolType` as shown in Figure 4-35.

```
typedef enum
{
    NETWORK_PROTOCOL_IP = 0,
    NETWORK_PROTOCOL_IPV6,
    NETWORK_PROTOCOL_MOBILE_IP,
    ...
   ROUTING_PROTOCOL_AODV6,
   ROUTING_PROTOCOL_DYMO,
   ROUTING_PROTOCOL_DYMO6,
    ROUTING_PROTOCOL_MYPROTOCOL
} NetworkRoutingProtocolType;
```

FIGURE 4-35. Adding MYPROTOCOL to List of Network Layer Protocols

Note **Always add to the end of lists in header files.**

A routing administrative distance is assigned to each routing protocol, which determines its priority relative to other routing protocols. A protocol with a lower administrative distance has a higher priority. The administrative distances of all routing protocols are defined in the enumeration `NetworkRoutingAdminDistanceType` defined in network.h (see Section 4.4.3).

For our example protocol, add the entry `ROUTING_ADMIN_DISTANCE_MYPROTOCOL` to `NetworkRoutingAdminDistanceType` as shown in Figure 4-36. Add this entry in the proper place in the list to reflect the desired priority of MYPROTOCOL relative to the other routing protocols.

```
typedef enum
{
    ROUTING_ADMIN_DISTANCE_STATIC = 1,
    ROUTING_ADMIN_DISTANCE_EBGPv4 = 20,
    ...
    ROUTING_ADMIN_DISTANCE_OLSR,
    ROUTING_ADMIN_DISTANCE_EIGRP,
    ROUTING_ADMIN_DISTANCE_MYPROTOCOL,
    //StartRIP
    ROUTING_ADMIN_DISTANCE_RIP,
    //EndRIP
   ...
   // Should always have the highest adminstrative distance
   // (ie, least important).
    ROUTING_ADMIN_DISTANCE_DEFAULT = 255
} NetworkRoutingAdminDistanceType;
```

FIGURE 4-36. Declaring Administrative Distance for MYPROTOCOL

4.2.6.4 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of an Application Layer routing protocol.

4.2.6.4.1 Determining the Protocol Configuration Format

A routing protocol may use protocol-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a routing protocol's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

As an example, the following are some of the configuration parameters for the Fisheye protocol. Refer to file EXATA_HOME/scenarios/default/default.config for an explanation of these parameters.

```
FISHEYE-INTRA-UPDATE-INTERVAL   5S
FISHEYE-INTER-UPDATE-INTERVAL   15S
FISHEYE-NEIGHBOR-TIMEOUT-INTERVAL 15S
```

A configuration variable is not always mandatory. If an optional configuration variable is not assigned a value, the default value is used. For example, if a user does not specify a value for `FISHEYE-INTRA-UPDATE-INTERVAL`, the default value of 5 seconds is used by the protocol.

4.2.6.4.2 Calling the Protocol Initialization Function

The initialization function of an Application Layer routing protocol is called from the Application Layer initialization function APP_Initialize, unlike the initialization function of a traffic-generating protocol which is called from function APP_InitializeApplications.

IP initialization function NetworkIpInit calls function NetworkIpParseAndSetRoutingProtocolType to read the name of the routing protocol for each interface from the configuration file and update the routing protocol type for that interface. Function APP_Initialize reads the name of the routing protocol from the configuration file, checks the routing protocol type for each interface, and updates the routing protocol information for that interface. The code snippet from NetworkIpParseAndSetRoutingProtocolType and APP_Initialize corresponding to the Bellman-Ford routing protocol are shown in Figure 4-37 and Figure 4-38, respectively. The functions used in these code snippets are explained below.

- Function IO_ReadString reads the name of the routing protocol from the configuration file. The prototype for IO_ReadString is defined in EXATA_HOME/include/fileio.h.
- Function NetworkIpGetInterfaceAddress, defined in EXATA_HOME/libraries/developer/src/network_ip.cpp, returns the IP address associated with an interface.

- Function NetworkIpAddUnicastRoutingProtocolType, defined in network_ip.cpp, initializes the routing protocol information for an interface. In the example of Figure 4-37, NetworkIpAddUnicastRoutingProtocolType updates the interface information to indicate that Bellman-Ford is the routing protocol running at that interface.
- Function RoutingBellmanfordInit, defined in routing_bellmanford.cpp, is the initialization function for Bellman-Ford. RoutingBellmanfordInit is called if the `bellmanford` field of `appdata` is `NULL` (see Section 4.2.3). RoutingBellmanfordInit creates an instance of the Bellman-Ford data structure and updates `bellmanford` to point to that data structure. Thus, RoutingBellmanfordInit is called at most once for each node, even if Bellman-Ford is running on multiple interfaces.

Figure 4-37 shows the modifications to be made to NetworkIpParseAndSetRoutingProtocolType and Figure 4-38 shows the modifications to be made to APP_Initialize to incorporate MYPROTOCOL in EXata. RoutingMyprotocolInit is the initialization function for MYPROTOCOL (see Section 4.2.6.4.3).

```
void
NetworkIpParseAndSetRoutingProtocolType (Node *node,
                                                    const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    BOOL retVal;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
         ...
         IO_ReadString(
                node->nodeId,
                NetworkIpGetInterfaceAddress(node, i),
                nodeInput,
                "ROUTING-PROTOCOL",
                &retVal,
                protocolString);
        if (retVal)
        {
            ...
            else if (strcmp(protocolString, "BELLMANFORD ") == 0)
            {
                    routingProtocolType = ROUTING_PROTOCOL_BELLMANFORD;
            }
            else if (strcmp(protocolString, "MYPROTOCOL ") == 0)
            {
                    routingProtocolType = ROUTING_PROTOCOL_MYPROTOCOL
            }
            ...
        }
        NetworkIpAddUnicastRoutingProtocolType(
                        node,
                        routingProtocolType,
                        i,
                    NETWORK_IPV4);
     }
     ...
}
```

FIGURE 4-37. Initializing Routing Protocol Information for an Interface

```
void
APP_Initialize(Node *node, const NodeInput *nodeInput)
{
    BOOL retVal;
    char buf[MAX_STRING_LENGTH];
    int i;
    ...
    node->appData.nextPortNum = 1024;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
     {
        NetworkType InterfaceType = NetworkIpGetInterfaceType(node,i);

         if (InterfaceType == NETWORK_IPV4 ||
            InterfaceType == NETWORK_DUAL)
         {
           switch (ip->interfaceInfo[i]->routingProtocolType)
           {
              case ROUTING_PROTOCOL_BELLMANFORD:
              {
                 if (node->appData.bellmanford == NULL)
                 {
                   RoutingBellmanfordInit(node);
                   RoutingBellmanfordInitTrace(node, nodeInput);
                 }
                 break;
              }
              case ROUTING_PROTOCOL_MYPROTOCOL:
              {
                 if (node->appData.myprotocol == NULL)
                 {
                    RoutingMyProtocolInit(node);
                    RoutingMyProtocolInitTrace(node, nodeInput);
                 }
                 break;
              }
              ...
           }
           ...
        }
        ...
     }
     ...
}
```

FIGURE 4-38. Calling Routing Protocol Initialization Function from APP_Initialize

4.2.6.4.3 Implementing the Protocol Initialization Function

The initialization of an Application Layer routing protocol takes place in the initialization function of the protocol that is called by the Application Layer initialization function APP_Initialize (see Figure 4-38). The initialization function of a routing protocol commonly performs the following tasks:

- Create an instance of the protocol data structure
- Read and store the user-specified configuration parameters
- Initialize the state variables and routing table
- Schedule a timer to itself for starting the protocol

Like all other functions belonging to the protocol, the prototype for the initialization function, RoutingMyprotocolInit, should be included in the protocol's header file, routing_myprotocol.h.

4.2.6.4.3.1 Creating an Instance and Reading Configuration Parameters

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as flags, connection information, routing table used by the protocol, etc.

To store the state, declare the structure to hold the protocol state in the header file, routing_myprotocol.h.

Create an instance of the protocol state by allocating memory to the state structure. Bellman-Ford performs this task in its initialization function RoutingBellmanfordInit by calling the function MEM_malloc to allocate memory for the Bellman-Ford data structure `bellmanford`, as shown in Figure 4-39. Refer to files routing_bellmanford.h and routing_bellmanford.cpp in EXATA_HOME/libraries/developer/src for details.

```
void RoutingBellmanfordInit(Node *node)
{
    Bellmanford *bellmanford;
    ...

    bellmanford = (Bellmanford *)
                    MEM_malloc(sizeof(Bellmanford));
    node->appData.bellmanford = (void *) bellmanford;
    ...
}
```

FIGURE 4-39. Creating Routing Protocol Instance in Initialization Function

The next step is to read the user-defined configuration parameters from the input file and store them in the protocol data structure. Since the EXata implementation of Bellman-Ford does not have any configurable parameters, we use Fisheye as an example. The Fisheye initialization function RoutingFisheyeInit, shown in Figure 4-40, uses the IO functions such as IO_ReadString to read parameter values from the input file and store them in the appropriate fields of the Fisheye protocol data structure `FisheyeData`. If a value is not specified for a parameter in the input file, RoutingFisheyeInit stores the default value for that parameter. Function Time_ConvertToClock, defined in EXATA_HOME/include/clock.h, converts a string to a EXata clocktype value. Refer to files routing_fisheye.h and routing_fisheye.cpp in EXATA_HOME/libraries/wireless/src for details. IO_ReadTime and IO_ReadString and other IO functions are defined in EXATA_HOME/include/fileio.h.

```
void
RoutingFisheyeInit(Node* node, const NodeInput* nodeInput)
{
    FisheyeData* fisheye;
    clocktype randomDelay;
    char buf[MAX_STRING_LENGTH];
    BOOL wasFound;
    int scope;

    ...
    IO_ReadString(node->nodeId,
                  NetworkIpGetInterfaceAddress(node, 0)
                  nodeInput,
                  "FISHEYE-INTRA-UPDATE-INTERVAL",
                  &wasFound,
                  buf);

    if (!wasFound) {
        fisheye->parameter.intraUpdateInterval =
            FISHEYE_INTRA_UPDATE_INTERVAL;
    }
    else {
        fisheye->parameter.intraUpdateInterval =
            TIME_ConvertToClock(buf);
    }

    IO_ReadString(node->nodeId,
                  NetworkIpGetInterfaceAddress(node, 0)
                  nodeInput,
                  "FISHEYE-INTER-UPDATE-INTERVAL",
                  &wasFound,
                  buf);

    if (!wasFound) {
        fisheye->parameter.interUpdateInterval =
            FISHEYE_INTER_UPDATE_INTERVAL;
    }
    else {
        fisheye->parameter.interUpdateInterval =
            TIME_ConvertToClock(buf);
    }
    ...
}
```

FIGURE 4-40. Reading Configurable Parameters

4.2.6.4.3.2 Initializing Timers

This is done in the same manner as for a traffic-generating protocol (see Section 4.2.5.5.3.3).

4.2.6.4.3.3 Initializing Tables

Most routing protocols initialize their routing tables in the initialization function. This may include allocating memory to the data structure and setting initial values as required. There is not one standard way of implementing a routing table and protocol designers are free to use any data structure well suited to their protocol.

Another commonly performed task in the initialization function is to add directly connected networks as permanent routes to the routing table. This can be performed by looping through all the interfaces of the node and writing its network address, interface address, and subnet mask to the routing table.

The code snippet from function RoutingBellmanfordInit, shown in Figure 4-41, demonstrates how to add directly connected networks to the routing table.

```
void RoutingBellmanfordInit(Node *node)
{
    Bellmanford *bellmanford;
    int i;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
        NodeAddress destAddress;
        NodeAddress subnetMask;
        Route *rowPtr;
        if (NetworkIpGetInterfaceType(node, i) != NETWORK_IPV4
            && NetworkIpGetInterfaceType(node, i) != NETWORK_DUAL)
        {
            continue;
        }
        if (NetworkIpIsWiredNetwork(node, i))
        {
            // This is a wiredlink interface.
           destAddress =
               NetworkIpGetInterfaceNetworkAddress(node, i);
           subnetMask =
               NetworkIpGetInterfaceSubnetMask(node, i);
         }
         else
         {
            // This is a wireless interface.
            destAddress =
                NetworkIpGetInterfaceAddress(node, i);
            subnetMask = ANY_DEST;
        }
        if (!FindRoute(bellmanford, destAddress))
         {
            rowPtr = AddRoute(bellmanford,
                        destAddress,
                        subnetMask,
                        NetworkIpGetInterfaceAddress(node, i),
                        i,
                        0);
            rowPtr->localRoute = TRUE;
         }
         ...
    }
    ...
}
```

FIGURE 4-41. Adding Directly Connected Networks to the Routing Table

4.2.6.5 Integrating with the Network Layer

All routing protocols, including those running at the Application Layer, interact with the IP protocol at the Network Layer. When a routing protocol is added at the Application Layer, the IP function NetworkRoutingGetAdminDistance needs to be modified.

The IP function NetworkRoutingGetAdminDistance, implemented in network_ip.cpp, returns the administrative distance of a routing protocol (see Section 4.2.6.3). Figure 4-42 shows the modifications that need to be made to NetworkRoutingGetAdminDistance to add MYPROTOCOL.

```
NetworkRoutingAdminDistanceType NetworkRoutingGetAdminDistance(
                                        Node *node,
                                        NetworkRoutingProtocolType type)
{
    switch (type)
    {
        case ROUTING_PROTOCOL_STATIC:
        {
            return ROUTING_ADMIN_DISTANCE_STATIC;
        }
        ...
        case ROUTING_PROTOCOL_BELLMANFORD:
        {
            return ROUTING_ADMIN_DISTANCE_BELLMANFORD;
        }
        case ROUTING_PROTOCOL_MYPROTOCOL:
        {
            return ROUTING_ADMIN_DISTANCE_MYPROTOCOL;
        }
        ...
    }
}
```

FIGURE 4-42. Modifications to Function NetworkRoutingGetAdminDistance

4.2.6.6 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for an Application layer routing protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the Application Layer, NODE_ProcessEvent calls the Application Layer event dispatcher APP_ProcessEvent, defined in application.cpp.

Section 4.2.6.6.1 describes how to modify the Application Layer dispatcher function to call the routing protocol's event dispatcher. Section 4.2.6.6.2 describes how to implement the event dispatcher for the routing protocol.

4.2.6.6.1 Modifying the Application Layer Event Dispatcher

Function APP_ProcessEvent implements the Application Layer event dispatcher that informs the appropriate Application Layer protocol of received events. Modify APP_ProcessEvent to call the routing protocol's event dispatcher when messages for the protocol are received. This modification is the similar to the modification made for a traffic-generating protocol (see Section 4.2.5.6.1), except that there is only one module for a routing protocol. Figure 4-43 shows the modifications that need to be made to APP_ProcessEvent to call MYPROTOCOL's event dispatcher function, RoutingMyprotocolLayer

```
void APP_ProcessEvent(Node *node, Message *msg)
{
    short protocolType;
    protocolType = APP_GetProtocolType(node,msg);
    switch(protocolType)
    {
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
        ...
        case APP_ROUTING_MYPROTOCOL:
        {
            RoutingMyprotocolLayer(node, msg);
            break;
        }
        ...
 }//switch//
}
```

FIGURE 4-43. Application Layer Event Dispatcher Function

4.2.6.6.2 Implementing the Routing Protocol Event Dispatcher

A protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received. A routing protocol typically handles two types of events: timers and packet events indicating reception of a *routing packet* from the Transport Layer. (Routing packets are control packets that carry information for the routing protocol. These routing packets are different from *data packets*, which carry user data.)

All event types used by EXata protocols are enumerated in the file api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as described in Section 4.2.5.6.2.

Figure 4-44 shows the event dispatcher function for Bellman-Ford, RoutingBellmanfordLayer, which is implemented in routing_bellmanford.cpp. `MSG_APP_PeriodicUpdateAlarm`, `MSG_APP_CheckRouteTimeoutAlarm`, and `MSG_APP_TriggeredUpdateAlarm` are timer events, and event `MSG_APP_FromTransport` indicates arrival of a routing packet from the Transport Layer. Actions taken in response to these events include: updating the IP forwarding table, broadcasting routing information, resetting timers, etc. Look at the implementation for Bellman-Ford or other Application Layer routing protocols to understand how such a protocol works.

After the message is handled by the event dispatcher, it frees the memory associated with the message by calling the function MESSAGE_Free.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

The event dispatcher also includes a default case in the switch statement to handle messages which contain an undefined event type.

```
void
RoutingBellmanfordLayer(Node *node, Message *msg)
{
    ...
    switch(msg->eventType)
    {
        // Messages sent within Bellman-Ford.

        case MSG_APP_PeriodicUpdateAlarm:
        {
            HandlePeriodicUpdateAlarm(node);
            break;
        }
        case MSG_APP_CheckRouteTimeoutAlarm:
        {
            HandleCheckRouteTimeoutAlarm(node);
            break;
        }
        case MSG_APP_TriggeredUpdateAlarm:
        {
            HandleTriggeredUpdateAlarm(node);
            break;
        }

        // Messages sent by UDP to Bellman-Ford.

        case MSG_APP_FromTransport:
        {
            HandleFromTransport(node, msg);
            break;
        }
        default:
            ERROR_ReportError("Invalid switch value");
    }

    // Done with the message, so free it.

    MESSAGE_Free(node, msg);
}
```

FIGURE 4-44. Event Dispatcher for Bellman-Ford

An Application Layer routing protocol cooperates with IP to perform the routing function. The routing protocol computes the routing information using its routing algorithm, and passes this routing information to IP. This routing information is stored in the IP forwarding table and is used by IP to route packets. Details of computing routing information are protocol-specific, but all Application Layer routing protocols use the following two functions, which are implemented in network_ip.cpp, to maintain the IP forwarding table:

1. NetworkEmptyFowardingTable: This function removes all entries in the forwarding table corresponding to a given routing protocol.
2. NetworkUpdateForwardingTable: This function updates an existing entry or adds a new entry to the forwarding table.

4.2.6.7 Collecting and Reporting Statistics

This step is similar to the one for adding a traffic-generating Application Layer protocol (see Section 4.2.5.7).

4.2.6.8 Finalization

The finalization function of the protocol is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

4.2.6.8.1 Modifying the Application Layer Finalization Function

This step is similar to the one in Section 4.2.5.8.1, except that in the Application Layer finalization function, APP_Finalize, the finalization functions of Application Layer routing protocols are called in the first switch statement and the second switch statement is used to call finalization functions of Application Layer traffic generators.

Figure 4-45 shows the modifications that need to be made to APP_finalize to call MYPROTOCOL's finalization function, RoutingMyprotocolFinalize.

```
void APP_Finalize(Node *node)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    ...
    int i;
    NetworkRoutingProtocolType routingProtocolType;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
        if (ip->interfaceInfo[i]->interfaceType == NETWORK_IPV4
            || ip->interfaceInfo[i]->interfaceType == NETWORK_DUAL)
        {
            routingProtocolType = ip->interfaceInfo[i]->routingProtocolType;
        }
        ...
        // Select application-layer routing protocol model and finalize.
        switch (routingProtocolType)
        {
            case ROUTING_PROTOCOL_BELLMANFORD:
            {
                RoutingBellmanfordFinalize(node, i);
                break;
            }
            case ROUTING_PROTOCOL_MYPROTOCOL:
            {
                RoutingMyprotocolFinalize(node, i);
                break;
            }
            ...
        }
       ...
   }
   ...
}
```

FIGURE 4-45. Calling Finalization Function of a Routing Protocol

4.2.6.8.2 Implementing the Routing Protocol Finalization Function

The finalization function of a routing protocol is similar to that of a traffic-generating protocol (see Section 4.2.5.8.2). In the finalization function for the routing protocol, RoutingMyprotocolFinalize, call the function to print statistics if routing statistics collection is enabled (`appData.routingStats` is true). Figure 4-46 shows the finalization function for Bellman-Ford, where function PrintStats is called to print statistics for Bellman-Ford.

```
void
RoutingBellmanfordFinalize(Node *node, int interfaceIndex)
{
    Bellmanford *bellmanford = (Bellmanford *)node->appData.bellmanford;

    ...
    if (node->appData.routingStats == TRUE
        && bellmanford->statsPrinted == FALSE)
    {
        PrintStats(node);
        bellmanford->statsPrinted = TRUE;
    }
}
```

FIGURE 4-46. Finalization Function of a Routing Protocol

4.2.6.9 Including and Compiling Files

This step is similar to the one for adding a traffic-generating Application Layer protocol (see Section 4.2.5.9).

4.2.7 Special Issues for Application Layer Protocols

4.2.7.1 Port Numbers In EXata

Just as every IP packet has a source and destination IP address, we must also specify a source and destination port number in every packet. Port numbers have to be in the range of 0-65535. However, known ports are almost always in the range 0-1023.

There are two types of port numbers to consider: client port numbers and server port numbers.

The client port number can be any port number at or above 1024. A client application may use a different port number each time it connects to a server. The field `nextPortNum` of the data structure `AppData` stores the next port number available at the node. This field is initialized to 1024 in the initialization function APP_Initialize (see Figure 4-38).

Most applications in EXata choose `nextPortNum` as the client port number. Each time a client instance is created, it is assigned `nextPortNum` as its port number and `nextPortNum` is incremented. Figure 4-20 shows how this is done for the CBR application. Assigning client port numbers in this manner ensures that each client instance gets a unique port number.

The client needs to know the server's port number in order to connect to the server. To enable the client to know it's port number the server uses a known port number which may have been agreed upon as a standard. All EXata application servers use as their known port number the value that is assigned to their application name in the enumerated data type `AppType` (see Section 4.2.3). For example, FTP_SERVER uses port number 21 as the destination port at the server.

The API functions for sending data using UDP at the Transport Layer accept the `AppType` of the server as an input parameter and set the destination port to this value. Figure 4-47 shows how this is done in the function APP_UdpSendNewdataWithPriority. (This function is implemented in EXATA_HOME/main/app_util.cpp.)

```
APP_UdpSendNewDataWithPriority(
    Node *node,
    AppType appType,
    NodeAddress sourceAddr,
    short sourcePort,
    NodeAddress destAddr,
    int outgoingInterface,
    char *payload,
    int payloadSize,
    TosType priority,
    clocktype delay,
    TraceProtocolType traceProtocol)
{
    Message *msg;
    AppToUdpSend *info;
    ActionData acnData;

    msg = MESSAGE_Alloc(
              node,
              TRANSPORT_LAYER,
              TransportProtocol_UDP,
              MSG_TRANSPORT_FromAppSend);
    MESSAGE_PacketAlloc(node, msg, payloadSize, traceProtocol);
    memcpy(MESSAGE_ReturnPacket(msg), payload, payloadSize);
    MESSAGE_InfoAlloc(node, msg, sizeof(AppToUdpSend));
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);
    SetIPv4AddressInfo(&info->sourceAddr, sourceAddr);
    info->sourcePort = sourcePort;
    SetIPv4AddressInfo(&info->destAddr, destAddr);
    info->destPort = (short) appType;
    info->priority = priority;
    info->outgoingInterface = outgoingInterface;

    //Trace Information
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node, msg, TRACE_APPLICATION_LAYER,
                     PACKET_OUT, &acnData);
    MESSAGE_Send(node, msg, delay);
}
```

FIGURE 4-47. Setting Port Numbers in API Calls

4.2.7.1.1 Overriding AppType as Destination Port

While it is recommended to use AppType as the destination port, it is possible to override this assignment and make a connection to a different port number. Though port numbers do not alter the statistics of an experiment, an application may need to use a specific port to facilitate tracing or for some special post processing scripts.

EXata provides several API function to enable overriding `AppType` as a destination port. Some of these are listed in Table 4-5. These APIs accept the desired destination port number as a parameter. Note that the first two functions listed in the table are overloaded functions. These functions are implemented in app_util.cpp.

TABLE 4-5. Examples of API Functions to Override Default Port Numbers

Function	Description
AppInfo * APP_RegisterNewApp(Node *node, AppType appType, void *dataPtr, short myPort)	Registers the application. Additionally inserts the port number (myPort) in the PortTable.
void APP_UdpSendNewHeaderVirtualDataWithPriority(Node *node, NodeAddress sourceAddr, short sourcePort, NodeAddress destAddr, short destinationPort, char *header int headerSize, int payloadSize, TosType priority, clocktype delay, TraceProtocolType traceProtocol)	Allocates header and virtual data with specified priority and sends to UDP. Delivers data to a particular destination port at the destination node. The destination port need not be the same as the AppType value.
void APP_TcpOpenConnectionWithPriority(Node *node, AppType appType, NodeAddress localAddr, short localPort, NodeAddress remoteAddr, short remotePort, int uniqueId, clocktype waitTime, TosType priority)	Opens a TCP connection with a specified priority and to a specified server port, which need not be the same as the AppType value.
void APP_TcpServerListen(Node *node, AppType appType, NodeAddress serverAddr, short serverPort)	Listens on a specified server port which need not be the same as the AppType value.

EXata provides several functions to manage port number assignment. These functions are listed in Table 4-6 and are implemented in app_util.cpp.

TABLE 4-6. Functions to Support Overriding of Default Port Numbers

Function	Description
short APP_GetFreePort(Node *node)	Returns a free port.
BOOL APP_IsFreePort(Node* node, unsigned short portNumber)	Checks if the port number is free or in use. Also checks if there is an application running at the node that uses an AppType that has been assigned the same value as this port number. This is done since applications may use AppType as destination port.
void APP_InserInPortTable(Node* node, AppType appType, unsigned short myPort)	Inserts information about the port number and the application using it in the port table. This function is called when registering an application at a new port by using APP_RegisterNewApp.

4.2.7.2 Setting Address for Broadcast Messages

For broadcasting packets, the destination address can be set to `ANY_DEST`, or the library function NetworkIpGetInterfaceBroadcastAddress can be used to get the interface broadcast address. `ANY_DEST` is a constant defined in EXATA_HOME/include/main.h and stands for any destination. Function NetworkIpGetInterfaceBroadcastAddress is defined in network_ip.h, and returns the broadcast address of the specified interface.

Figure 4-48 shows a code snippet from the Bellman-Ford function SendRouteAdvertisement (implemented in file routing_bellmanford.cpp) that sets the destination address to the interface broadcast address for the wired interface and to `ANY_DEST` for the wireless interface.

```
static void SendRouteAdvertisement(Node *node, RouteAdvertisementType type)
{
  ...
  int i;
  for (i = 0; i < node->numberInterfaces; i++)
    {
      NodeAddress destAddress;
      ...
      if (NetworkIpIsWiredNetwork (node, i))
      {
          destAddress = NetworkIpGetInterfaceBroadcastAddress(node, i);
      }
     else
      {
          destAddress = ANY_DEST;
      }
   ...
    }
  ...
}
```

FIGURE 4-48. Setting Broadcast Address

4.3 Transport Layer

The Transport Layer resides between the Application and Network Layers in the EXata protocol stack, as shown in Figure 4-1. The Transport Layer provides the service of transporting Application Layer data between the client and the server of an application. The Transport Layer uses Network Layer services to provide a data delivery service to the Application Layer. The Transport Layer provides services for both connection-mode transmissions and for connectionless-mode transmissions.

When the Application Layer sends a data packet to the Transport Layer, the Transport Layer header is appended to the packet, and the packet is passed down to the next layer, the Network Layer. Similarly, when the Transport Layer receives a packet from the Network Layer, it removes the Transport Layer header from the data packet and sends the packet up to the Application Layer.

This section gives a detailed description of how to add a Transport Layer protocol to EXata.

4.3.1 Transport Layer Protocols in EXata

The Transport Layer of the TCP/IP protocol suite consists of two protocols, UDP and TCP. UDP provides an unreliable connectionless delivery service to send and receive messages. TCP provides a reliable delivery service on top of the IP datagram delivery service. EXata also supports the Multicast Dissemination Protocol (MDP), which is a reliable multicast transport protocol built on top of the generic UDP/IP multicast transport. EXata also provides an extension of RSVP (RSVP-TE) that is used to Label Switch Path (LSP) determination for MPLS (Multiprotocol Label Switching).

See *Developer Model Library* for a description of EXata's MDP, UDP, and TCP models and the MPLS section of *Multimedia and Enterprise Model Library* for a description of EXata's RSVP-TE model.

4.3.1.1 Multicast Dissemination Protocol (MDP)

The Multicast Dissemination Protocol (MDP) is a transport protocol which allows data to be reliably transmitted from a source (server) to a set of receivers on top of the generic UDP/IP multicast transport. MDP is well suited for reliable multicast bulk transfer of data across a heterogeneous inter-network. At its core, it is an efficient negative acknowledgement (NACK) based reliable multicast protocol and also includes an optional adaptive end-to-end rate-based congestion control mode.

4.3.1.2 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) offers only a minimal transport service, i.e., an unreliable datagram delivery. UDP is a connectionless protocol, and, hence, does not incur the delay associated with connection establishment. UDP also has a smaller header than TCP. Lastly, UDP does not implement any congestion control mechanism, which means that although some packets may be lost, the packets that are delivered incur minimum transmission delay. These features make UDP suitable for certain applications.

In EXata, the CBR (constant bit rate) is an example of an application that uses UDP at the Transport Layer.

4.3.1.3 Transmission Control Protocol (TCP)

TCP, Transmission Control Protocol, is a Transport Layer connection-oriented byte stream protocol. This protocol is typically used by applications that require guaranteed delivery. To provide a reliable transport service, TCP implements a connection management mechanism, maintains the connection state in the end systems, and implements a congestion control mechanism.

TCP Variants in EXata

EXata TCP implementation supports the following TCP variants:

- Abstract TCP: Abstract TCP is based on TCP Reno. It omits some features oF TCP Reno to improve runtime performance at the cost of reduced fidelity.
- TAHOE: This TCP variant implements three TCP congestion control algorithms, namely Slow Start, Congestion Avoidance, and Fast Retransmit.
- RENO: This TCP variant extends TAHOE to include a fourth congestion control algorithm called Fast Recovery.
- LITE: This TCP variant extends RENO by including two additional options, namely Big Window and Protection Against Wrapped Sequence Numbers.
- NEWRENO: This TCP variant is the same as RENO except for a modification to Fast Recovery (i.e., an ACK for highest sequence number sent must be received to exit Fast Recovery).
- SACK: This TCP variant is an extension of RENO that includes Selective Acknowledgements and Selective Retransmissions.
- Abstract: This TCP variant is based on RENO. It simplifies and omits some features to improve the performance.

Examples of TCP-based applications implemented in EXata are FTP, FTP/GEN and HTTP.

4.3.1.4 Reservation Protocol with Traffic Engineering (RSVP-TE)

Reservation Protocol (RSVP) is a control protocol in the Transport Layer. This protocol is not used to transmit data from one node to another. This is only used as a control protocol in the Transport Layer and determines the resources in a path from a source to a destination for the quality of service requested. At present, EXata does not support the basic RSVP service but implements an extension of RSVP called Reservation Protocol with Traffic Engineering (RSVP-TE). RSVP-TE is an extension of RSVP for establishing Label Switched Paths (LSPs) in Multiprotocol Switching Paths (MPLS) networks, with or without resource reservations. When a LSP is established between two MPLS nodes, packets can be delivered through this LSP tunnel without considering IP Layer routing techniques.

4.3.2 Transport Layer Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to Transport Layer protocols. These files contain detailed comments on functions and other code components.

The Transport Layer API is composed of several macros, functions, and structures. These are defined in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/transport.h

  This file contains definitions common to Transport Layer protocols and transport data structure in the node structure.

Additionally, the following header file is also relevant to the Transport Layer:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.

The following are the folders and source files associated with the Transport Layer:

- EXATA_HOME/main/transport.cpp
  This file contains Transport Layer functions, including the initialization, message processing, and finalization functions.
- EXATA_HOME/libraries/developer/src
  This folder contains the source and header files for the UDP protocol and the various TCP versions.
- EXATA_HOME/libraries/multimedia_enterprise/src
  This folder contains the source and header files the RSVP-TE implementation in EXata.

4.3.3 Transport Layer Data Structures

The Transport Layer data structures are defined in EXATA_HOME/include/transport.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file transport.h for a complete description.)

1. `TransportProtocol`: This is an enumeration type that lists all the Transport Layer protocols.

```
typedef enum {
   TransportProtocol_UDP,
   TransportProtocol_TCP,
   TransportProtocol_RSVP
} TransportProtocol;
```

2. `TransportData`: This is the main data structure used by the Transport Layer and stores information about all protocols running at the Transport Layer. It also stores a Boolean variable for each non-mandatory protocol to indicate whether the protocol is enabled or disabled in the configuration.

```
struct TransportData
{
   TransportDataUdp* udp;
   int tcpType;
   void* tcp;
   BOOL rsvpProtocol;
   void *rsvpVariable;
}
```

   - `udp`: Pointer the data structure for the UDP protocol.
   - `tcpType`: Variable that indicates the TCP protocol type (TCP regular or TCP Abstract)
   - `tcp`: Pointer to the data structure for the TCP protocol.
   - `rsvpProtocol`: Flag that indicates whether the RSVP-TE protocol is enabled.
   - `rsvpVariable`: Pointer to the data structure for the RSVP-TE protocol.

4.3.4 Transport Layer APIs and Inter-layer Communication

This section describes the APIs used by the Application Layer to communicate with the Transport Layer (see Section 4.3.4.1), message types that are used by the Transport Layer to communicate with the Application Layer (see Section 4.3.4.2), the API used by Transport Layer protocols to communicate with the Network Layer (see Section 4.3.4.3), and the message type used by the Network Layer to communicate with the Transport Layer (see Section 4.3.4.4).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.3.4.1 Application Layer to Transport Layer Communication

Application Layer protocols use several APIs to communicate with the Transport Layer (see Section 4.2.4.1). The file EXATA_HOME/main/app_util.cpp contains the implementation of these functions. These APIs are implemented using messages. The message types used in the implementation of these APIs are enumerated in the file EXATA_HOME/include/api.h. Some of the message types used by the Application Layer to communicate with the Transport Layer are listed below.

- `MSG_TRANSPORT_FromAppSend`: This message type is used by the Application Layer to send data to the Transport Layer.
- `MSG_TRANSPORT_FromAppListen`: This message type is used by the Application Layer to direct TCP to listen on a port.
- `MSG_TRANSPORT_FromAppOpen`: This message type is used by the Application Layer to request TCP to open a TCP connection.
- `MSG_TRANSPORT_FromAppClose`: This message type is used by the Application Layer to request TCP to close a TCP connection.

4.3.4.2 Transport Layer to Application Layer Communication

Transport Layer protocols communicate with the Application Layer by means of messages. The message types used for this communication are enumerated in the file api.h. Some of the message types used by Transport Layer protocols to communicate with the Application Layer are listed below.

- `MSG_APP_FromTransport`: This message type is used by UDP to pass an incoming packet to the Application Layer.
- `MSG_APP_FromTransOpenResult`: This message type is used by TCP to notify an application client that a TCP connection request was accepted or rejected.
- `MSG_APP_FromTransDataSent`: This message type is used by TCP to indicate to the Application Layer that an outgoing packet has been transmitted.
- `MSG_APP_FromTransDataReceived`: This message type is used by TCP to pass an incoming packet to the Application Layer.
- `MSG_APP_FromTransListenResult`: This message type is used by TCP to notify an application server that a request to open a TCP connection has been received.
- `MSG_APP_FromTransCloseResult`: This message type is used by TCP to notify an application client or server that a TCP connection has been closed.

4.3.4.3 Transport Layer to Network Layer Communication

The Transport Layer communicates with the Network Layer by using the API NetworkIpReceivePacketFromTransportLayer. This function sends a packet from a Transport Layer protocol (UDP, TCP or RSVP-TE) to the IP protocol at the Network Layer. The prototype for this function is contained in the file EXATA_HOME/libraries/developer/src/network_ip.h. The file EXATA_HOME/libraries/developer/src/network_ip.cpp contains the implementation of NetworkIpReceivePacketFromTransportLayer.

4.3.4.4 Network Layer to Transport Layer Communication

The IP protocol at the Network Layer uses several APIs to communicate with the Transport Layer protocols: SendToUdp, SendToTcp, SendToRsvp, and SendToTransport (see Section 4.4.4.2). The prototype for this function is contained in the file network_ip.h. The file network_ip.cpp contains the implementation of these functions. These APIs are implemented using messages. These messages use the message type `MSG_TRANSPORT_FromNetwork`, which is used to pass incoming packets to Transport Layer protocols.

4.3.5 Adding a Transport Layer Protocol

This section provides an overview of the flow of a Transport Layer protocol and provides an outline for developing and adding a new Transport Layer protocol to EXata. It describes how to develop code components common to most transport protocols, such as initializing, sending and receiving packets, and collecting statistics.

We illustrate the process of adding a Transport Layer protocol by using as an example the implementation code for UDP. The header file for the UDP implementation is transport_udp.h and the source file is transport_udp.cpp in the folder EXATA_HOME/libraries/developer/src. We use code segments from these two files throughout this section to illustrate different steps in writing a transport protocol. After understanding the discussed snippets, look at the complete code for UDP to understand how a transport protocol is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a Transport Layer protocol to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.3.5.2).
2. Modify the file transport.cpp to include the protocol's header file (see Section 4.3.5.2).
3. Include the protocol in the list of Transport Layer protocols and trace protocols (see Section 4.3.5.3).
4. Define data structures for the protocol (see Section 4.3.5.4).
5. Decide on the format for the protocol-specific configuration parameters (see Section 4.3.5.5.1).
6. Read the protocol's configuration parameters and call the protocol's initialization function from the Transport Layer initialization function, TRANSPORT_Initialize (see Section 4.3.5.5.2).
7. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Declare and initialize the state variables (see Section 4.3.5.5.3.1).
   b. Initialize timers (see Section 4.3.5.5.3.2).
8. Call the protocol event dispatcher from the Transport Layer event dispatcher, TRANSPORT_ProcessEvent (see Section 4.3.5.6.1).
9. Declare any new event types used by the protocol in the header file api.h (see Section 4.3.5.6.2).
10. Write the protocol event dispatcher (see Section 4.3.5.6.2).
11. Integrate the protocol with the Network Layer (see Section 4.3.5.8).
    a. Define an IP Protocol Number for the protocol.
    b. Write a function to deliver packets from IP to the protocol.
    c. Call the function to deliver packets from IP to the protocol from the IP function DeliverPacket.
12. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.3.5.9.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.3.5.9.2).
    c. Update the statistics as appropriate (see Section 4.3.5.9.3).
    d. Write a function to print the statistics (see Section 4.3.5.9.4).
    e. Add dynamic statistics to the protocol, if desired (see Section 4.3.5.9.5).
13. Call the protocol finalization function from the Transport Layer finalization function, TRANSPORT_Finalize (see Section 4.3.5.10.1).
14. Write the protocol finalization function (see Section 4.3.5.10.2). Call the function to print statistics from the protocol finalization function.

15. Include the protocol header and source files in the EXata tree and compile (see Section 4.3.5.11).

16. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.3.5.12).

4.3.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new protocol, the programmer name the different components of the new protocol in a similar manner. It will be helpful to examine the implementation of UDP in EXata for hints for naming and coding different components of the new protocol.

In this section, we describe the steps for developing a Transport Layer protocol called "MYPROTOCOL". We will use the string "Myprotocol" in the names of the different components of this protocol, just as the string "Udp" appears in the names of the components of the UDP implementation.

4.3.5.2 Creating Files

The first step towards adding a transport protocol is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/developer/src. However, it is recommended that all user-developed models be made part of a separate library. In our example, we will place the transport model in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn't already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the transport model and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *transport_* to designate the files as transport model files.

Examples:

- transport_udp.cpp, transport_udp.h: Implement UDP (User Datagram Protocol)
- transport_rsvp.cpp, transport_rsvp.h: Implement RSVP-TE (Reservation Protocol with Traffic Engineering)

In keeping with the naming guidelines of Section 16, the header file for the example protocol is called transport_myprotocol.h, and the source file is called transport_myprotocol.cpp.

It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems, even if the compilation process does not indicate any error.

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, transport_myprotocol.h, should contain the following:

- Prototypes for interface functions in source file, transport_myprotocol.cpp
- Constant definitions
- Data structure definitions and data types: `struct` and `enum` declarations

The source file, transport_myprotocol.cpp, should contain the following:

- Statement to include the protocol's header file:

```
#include "transport_myprotocol.h"
```

- Statements to include standard library functions and other header files needed by the protocol source file. A typical protocol source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "api.h"
#include "partition.h"
#include "network_ip.h"
```

- Protocol initialization function, TransportMyprotocolInit
- Protocol event dispatcher function, TransportMyprotocolLayer
- Protocol finalization function, MyprotocolFinalize
- Additional protocol implementation functions

The file EXATA_HOME/main/transport.cpp contains the layer level initialization, event dispatcher, and finalization functions. These layer level functions in turn call the protocol functions TransportMyprotocolInit, TransportMyprotocolLayer, and MyprotocolFinalize. Therefore, to make these protocol functions available to the layer level functions, insert the following include statement in the file transport.cpp:

```
#include "transport_myprotocol.h"
```

4.3.5.3 Including MYPROTOCOL in List of Transport Protocols

Each node in EXata hosts an operating protocol stack. For each layer in the stack, a list of protocols running at that layer is maintained. When a new Transport Layer protocol is added to EXata, it needs to be included in the list of Transport Layer protocols. To do this, add the protocol name to the enumeration `TransportProtocol` defined in transport.h (see Section 4.3.3).

For our example protocol, add the entry `MYPROTOCOL` to `TransportProtocol` as shown in Figure 4-49.

```
typedef enum {
    TransportProtocol_UDP,
    TransportProtocol_TCP,
    TransportProtocol_RSVP,
    TransportProtocol_MYPROTOCOL
} TransportProtocol;
```

FIGURE 4-49. Adding MYPROTOCOL to List of Transport Layer Protocols

Note **Always add to the end of lists in header files.**

EXata provides for detailed traces of packets as they traverse the protocol stack at nodes in the network. A packet trace lists, among other information, the protocol that is handling the packet at the time of the trace. To facilitate tracing, EXata lists all protocols in an enumeration, `TraceProtocolType`, in the file EXATA_HOME/include/trace.h. For our example protocol, add an entry `TRACE_MYPROTOCOL` in `TraceProtocolType`, as shown in Figure 4-50.

```
typedef enum
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,                  // 1
    TRACE_UDP,                  // 2
    TRACE_IP,                   // 3
    TRACE_CBR,                  // 4
    ...

    TRACE_MYPROTCOL,
    // Must be last one!!!
    TRACE_ANY_PROTOCOL
}TraceProtocolType;
```

FIGURE 4-50. Adding MYPROTOCOL to List of Trace Protocols

Note **Always add to the end of lists in header files (just before the entry `TRACE_ANY_PROTOCOL`).**

The state information for each transport protocol is stored in `TransportData`. Modify `TransportData` to include a pointer to the new protocol's state. Also, if the new protocol is not mandatory, then include a Boolean flag to indicate whether the protocol is enabled. An example of adding a non-mandatory protocol is shown in Figure 4-51.

```
struct struct_transport_str
{
   TransportDataUdp* udp;
   int tcpType;
   void* tcp;
   BOOL rsvpProtocol;
   void *rsvpVariable;
   BOOL myprotocolEnabled;    // Flag to indicate if protocol is enabled
   void *myprotocolVariable   // Pointer to protocol's state
}
```

FIGURE 4-51. Adding MYPROTOCOL to TransportData

4.3.5.4 Defining Data Structures

Each Transport Layer protocol has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Protocol parameters (see Section 4.3.5.5.2)
2. Protocol state (see Section 4.3.5.5.3)
3. Statistics variables (see Section 4.3.5.9.1)

Define appropriate data structures for your protocol in the protocol header file, transport_myprotocol.h. As an example, the following data structure (defined in transport_udp.h) is used by the UDP protocol:

```
struct TransportDataUdpStruct {
    BOOL udpStatsEnabled;            /* whether to collect stats */
    TransportUdpStat *statistics;    /* statistics variable*/
    BOOL traceEnabled;
};
```

In the above declaration, `TransportUdpStat` is the data structure for UDP statistics (see Section 4.3.5.9.1).

4.3.5.5 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a Transport Layer protocol.

4.3.5.5.1 Determining the Protocol Configuration Format

A protocol may use protocol-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a protocol's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

As an example, the following configuration specifies the TCP variant to be used:

```
TCP    RENO
```

A configuration variable is not always mandatory. If an optional configuration variable is not assigned a value in the configuration file, the default value is used. For example, if a user does not specify the TCP variant in the configuration file, the default variant LITE will be used for TCP.

The new Transport Layer protocol may not be a mandatory service in the Transport Layer. In that case, use a configuration parameter to enable or disable the protocol. The syntax of this parameter is as follows:

```
TRANSPORT-PROTOCOL-<Protocol_name> <status>
```

where

`<Parameter-name>`	: Name of the protocol.
`<status>`	: YES, if the protocol is enabled; NO, if the protocol is disabled.

For our example protocol MYPROTOCOL, the following statement indicates that the protocol is enabled:

```
TRANSPORT-PROTOCOL-MYPROTOCOL YES
```

Define a configuration parameter for the new Transport Layer protocol which determines whether statistics are to be collected for the protocol. If this parameter is set to YES in the configuration file, statistics collection is enabled for the protocol; if the parameter is set to NO, statistics collection is disabled for the protocol. For our example protocol MYPROTOCOL, the following statement indicates that statistics collection is enabled:

```
MYPROTOCOL-STATISTICS YES
```

Decide on the format for specifying the new protocol's configuration parameters. Section 4.3.5.5.2 explains how to read user input specified in this format to initialize the protocol.

4.3.5.5.2 Reading Configuration Parameters and Calling the Protocol Initialization Function

EXata can configure a protocol to the parameters specified by the user in the EXata configuration file that sets up the experiment. This section explains how to read these user-specified configuration parameters for the protocol and provide them to the protocol's initialization function.

The protocol stack of each node is initialized in a bottom up manner. The initialization of the Transport Layer thus occurs after the layers below it have been initialized. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the Transport Layer initialization function TRANSPORT_Initialize, which is implemented in the file transport.cpp. Function TRANSPORT_Initialize reads the configuration parameters and stores them in the appropriate data structure, and calls the initialization routine for the protocol as follows:

- If the protocol is mandatory, such as UDP and TCP, TRANSPORT_Initialize calls its initialization function, as shown in Figure 4-52. TransportUdpInit, defined in transport_udp.cpp, is the initialization function for UDP.
- If the transport protocol is non-mandatory, such as RSVP-TE, then TRANSPORT_Initialize first finds whether the protocol is enabled by searching the keyword TRANSPORT-PROTOCOL-<protocol_name>. The Boolean flag corresponding to the protocol is set accordingly, and if the protocol is enabled, then its initialization function is called. Function IO_ReadString, defined in EXATA_HOME/include/fileio.h, is a utility function that retrieves from an input file the value associated with a string parameter.

For our example non-mandatory protocol, MYPROTOCOL, Figure 4-52 gives an example of code to be added to TRANSPORT_Initialize to call the initialization function for MYPROTOCOL. The Boolean variable myprotocolEnabled (see Section 4.3.5.3) is set to TRUE or FALSE depending upon the value read from the configuration file (see Section 4.3.5.5.1). Function TransportMyprotocolInit is the initialization function of MYPROTOCOL.

```
void TRANSPORT_Initialize(Node * node, const NodeInput * nodeInput)
{
    BOOL wasFound = FALSE;
    char buf[MAX_STRING_LENGTH];

    node->transportData.tcp = NULL;
    node->transportData.udp = NULL;

    TransportTcpInit(node, nodeInput);
    TransportUdpInit(node, nodeInput);
   ...

    /* Initialize MYPROTCOL. */

    node->transportData.myprotocolVariable = NULL;
    IO_ReadString(node->nodeId, ANY_ADDRESS, nodeInput,
                  "TRANSPORT-PROTOCOL-MYPROTOCOL", &wasFound, buf);

    if (wasFound)
    {
       if (strcmp(buf, "YES") == 0)
       {
            node->transportData.myprotocolEnabled = TRUE;
            TransportMyprotocolInit(node, nodeInput);
       }
       else
       if (strcmp(buf, "NO") == 0)
       {
          node->transportData.myprotocolEnabled = FALSE;
       }
       else
       {
          ERROR_ReportError("Expecting YES or NO for"
                            "TRANSPORT-PROTOCOL-MYPROTOCOL parameter\n");
       }
    }
    else  /* Not found */
    {
       node->transportData.myprotocolEnabled = FALSE;
    }
}
```

FIGURE 4-52. Calling the Protocol Initialization Function

4.3.5.5.3 Implementing the Protocol Initialization Function

The initialization of a Transport Layer protocol takes place in the initialization function of the protocol that is called by the Transport Layer initialization function TRANSPORT_Initialize. The initialization function of a protocol commonly performs the following tasks:

- Initialize the state and store the user specified configuration parameters
- Create an instance of the protocol
- Initialize data structures and variables as required, e.g., allocate memory to tables, set default values, etc.
- Schedule a timer to itself for starting the protocol

Like all other functions belonging to the protocol, the prototype for the initialization function should be included in the protocol's header file, transport_myprotocol.h.

4.3.5.5.3.1 Creating an Instance and Initializing the State

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as flags, connection information, sequence number, pointers to tables used by the protocol, etc.

To store the state, declare the structure to hold the protocol state in the header file, transport_myprotocol.h (see Section 4.3.5.4).

Create an instance of the protocol state by allocating memory to the state structure. UDP performs this task in its initialization function TransportUdpInit by calling the function MEM_alloc to allocate memory for the UDP data structure TransportDataUdp, as shown in Figure 4-53.

Update the Transport Layer data structure, TransportData, so that the field for the protocol points to the newly created instance of the protocol data structure, e.g., TransportInitUdp updates the udp field of TransportData to point to the newly created instance of the UDP data structure.

```
void
TransportUdpInit(Node *node, const NodeInput *nodeInput)
{
    char buf[MAX_STRING_LENGTH];
    BOOL retVal;

    TransportDataUdp* udp =
       (TransportDataUdp*)MEM_malloc(sizeof(TransportDataUdp));
    node->transportData.udp = udp;

    TransportUdpInitTrace(node, nodeInput);

    IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "UDP-STATISTICS",
        &retVal,
        buf);
    if (retVal == FALSE || strcmp(buf, "NO") == 0)
    {
        udp->udpStatsEnabled = FALSE;
    }
    else if (strcmp(buf, "YES") == 0)
    {
        udp->udpStatsEnabled = TRUE;
    }
    else
    {
       printf("TransportUdp unknown setting (%s) for UDP-STATISTICS.\n", buf);
        udp->udpStatsEnabled = FALSE;
    }

    if (udp->udpStatsEnabled == TRUE)
    {
        udp->statistics = (TransportUdpStat *)
            MEM_malloc(sizeof(TransportUdpStat));
        if (udp->statistics == NULL)
        {
            printf("TRANSPORT UDP: cannot allocate memory\n");
            abort();
        }
        memset(udp->statistics, 0, sizeof(TransportUdpStat));

        std::string path;
        D_Hierarchy *h = &node->partitionData->dynamicHierarchy;
        ...
    }
}
```

FIGURE 4-53. UDP Initialization Function

4.3.5.5.3.2 Initializing Timers

In addition to initializing data structures, the initialization function may also initialize timers for the transport protocol. Section 3.3.2.2 discusses in detail how to use timers.

4.3.5.6 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a transport protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the Transport Layer, NODE_ProcessEvent calls the Transport Layer event dispatcher TRANSPORT_ProcessEvent, defined in transport.cpp.

Section 4.3.5.6.1 describes how to modify the Transport Layer event dispatcher function to call the transport protocol's event dispatcher. Section 4.3.5.6.2 describes how to implement the protocol's event dispatcher.

4.3.5.6.1 Modifying the Transport Layer Event Dispatcher

Function TRANSPORT_ProcessEvent implements the Transport Layer event dispatcher that informs the appropriate transport protocol of received events. Messages contain the name of the protocol they are destined for. (This is the transport protocol name specified in the enumerated data type `TransportProtocol`, described in Section 4.3.3.) The API function MESSAGE_GetProtocol returns the protocol for which the message is destined. TRANSPORT_ProcessEvent implements a switch statement on the protocol name read from the message and calls the appropriate protocol-specific event dispatcher.

To enable the protocol MYPROTOCOL to receive events, add code to TRANSPORT_ProcessEvent to call the protocol's event dispatcher function when messages for the protocol are received. If MYPROTOCOL is a non-mandatory protocol, check if it is enabled before calling its event dispatcher function. Figure 4-54 shows a code fragment from TRANSPORT_ProcessEvent with sample code for calling MYPROTOCOL's event dispatcher function TransportMyprotocolLayer.

```
void
TRANSPORT_ProcessEvent(Node * node, Message * msg)
{
    switch (MESSAGE_GetProtocol(msg))
    {
        case TransportProtocol_UDP:
        {
            TransportUdpLayer(node, msg);
            break;
        }
        case TransportProtocol_TCP:
        {
            TransportTcpLayer(node, msg);
            break;
        }
        case TransportProtocol_RSVP:
        {
            ...
        }
        case TransportLayer_MYPROTOCOL:
        {
            if (node->transportData.myprotocolEnabled == FALSE)
            {
                ERROR_ReportError("MYPROTOCOL is not enabled\n");
             }
            TransportMyprotocolLayer(node, msg);
            break;
        }
        default:
            assert(FALSE); abort();
            break;
    }//switch//
}
```

FIGURE 4-54. Transport Layer Event Dispatcher

4.3.5.6.2 Implementing the Protocol Event Dispatcher

A protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received.

All event types used by EXata protocols are enumerated in the file api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as shown in Figure 4-55.

```
// /**
// ENUM        :: MESSAGE/EVENT
// DESCRIPTION :: Event/message types exchanged in the simulation
// **/
enum
{
    /* Special message types used for internal design. */
    MSG_SPECIAL_Timer                          = 0,
    ...
    /* Message Types for Channel layer */
    MSG_PROP_SignalArrival                     = 100,
    MSG_PROP_SignalEnd                         = 101,
    ...
    /*
     * Any other message types which have to be added should be added before
     * MSG_DEFAULT. Otherwise the program will not work correctly.
     */
    MSG_TRANSPORT_PROTOCOL_NewEvent1,
    MSG_TRANSPORT_PROTOCOL_NewEvent2,
    MSG_DEFAULT                                = 10000
};
```

FIGURE 4-55. Declaring New Event Types

Note **Always add to the end of lists in header files (just before the entry `MSG_DEFAULT`).**

The event dispatcher function for a protocol that provides a transport service, such as UDP and TCP, is different from the event dispatcher for a protocol that provides a supplemental or auxiliary service, such as RSVP-TE. In the former case, the protocol interacts with both Application and Network Layers, while in the latter case, the protocol interacts only with the Network Layer. Section 4.3.5.6.2.1 describes the event dispatcher for UDP and Section 4.3.5.6.2.2 describes the event dispatcher for RSVP-TE.

Write MYPROTOCOL's event dispatcher function, TransportMyprotocolLayer, using UDP or RSVP-TE as an example. Include the prototype for TransportMyprotocolLayer in the protocol's header file, transport_myprotocol.h.

4.3.5.6.2.1 UDP Event Dispatcher

UDP provides a transport service to Application Layer protocols. When UDP receives a packet from the Application Layer, it appends a UDP header to the packet and sends the packet to the Network Layer. When UDP receives a packet from the Network Layer, it removes the UDP header from the packet and sends the packet to the Application Layer. Function TransportUdpLayer, shown in Figure 4-56, is the event dispatcher for UDP. Event `MSG_TRANSPORT_FromNetwork` corresponds to receiving a packet from the Network Layer and event `MSG_TRANSPORT_FromAppSend` corresponds to receiving a packet from the Application Layer.

The event dispatcher also includes a default case in the switch statement to handle events of an unknown type.

See files transport_udp.h and transport_udp.cpp in EXATA_HOME/libraries/developer/src for definitions of data structures and functions used for implementing UDP.

```
void
TransportUdpLayer(Node *node,  Message *msg)
{
    switch (msg->eventType)
    {
        case MSG_TRANSPORT_FromNetwork:
        {
            TransportUdpSendToApp(node, msg);
            break;
        }
        case MSG_TRANSPORT_FromAppSend:
        {
            TransportUdpSendToNetwork(node, msg);
            break;
        }
        default:
            assert(FALSE);
            abort();
    }
}
```

FIGURE 4-56. Event Dispatcher for UDP

Function TransportUdpLayer calls function TransportUdpSendToApp when it receives a packet from the Network Layer. Function TransportUdpSendToApp, shown in Figure 4-57, performs the following actions:

1. Reads the source address, destination address and incoming interface index from the info field of the message.
2. Reads the destination port number from the UDP header (which is at the beginning of the packet field of the message), using the API MESSAGE_ReturnPacket.
3. Allocates a new info field for the message, using the API MESSAGE_InfoAlloc.
4. Copies the source address, destination address, incoming interface index and the destination port number into the new info field of the message.
5. Removes the header, using the API MESSAGE_RemoveHeader.
6. Schedules a packet receive event (`MSG_APP_FromTransport`) for the destination protocol at the Application Layer, using the APIs MESSAGE_SetLayer, MESSAGE_SetEvent and MESSAGE_Send.

```
void
TransportUdpSendToApp(Node *node, Message *msg)
{
    TransportDataUdp *udpLayer =
        (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader* udpHdr = (TransportUdpHeader *)
                                 MESSAGE_ReturnPacket(msg);
    UdpToAppRecv *info;
    Address sourceAddress;
    Address destinationAddress;
    NetworkToTransportInfo *infoPtr = (NetworkToTransportInfo *)
                                      MESSAGE_ReturnInfo(msg);
    int incomingInterfaceIndex = infoPtr->incomingInterfaceIndex;

    memcpy(&sourceAddress, &(infoPtr->sourceAddr), sizeof(Address));
    memcpy(&destinationAddress,
           &(infoPtr->destinationAddr),
           sizeof(Address));
    if (udpLayer->udpStatsEnabled == TRUE)
    {
        udpLayer->statistics->numPktToApp++;
    }
    /* Set destination port. */
    MESSAGE_SetLayer(msg, APP_LAYER, udpHdr->destPort);
    MESSAGE_SetEvent(msg, MSG_APP_FromTransport);
    MESSAGE_SetInstanceId(msg, 0);

    /* Update info field (used by application layer). */
    TosType_priority = infoPtr->priority;
    MESSAGE_InfoAlloc(node, msg, sizeof(UdpToAppRecv));
    info = (UdpToAppRecv *) MESSAGE_ReturnInfo(msg);

    info->priority = priority;
    memcpy(&(info->sourceAddr), &sourceAddress, sizeof(Address));
    info->sourcePort = udpHdr->sourcePort;
    memcpy(&(info->destAddr), &destinationAddress, sizeof(Address));
    info->destPort = udpHdr->destPort;
    info->incomingInterfaceIndex = incomingInterfaceIndex;

    ActionData acnData;
    acnData.actionType = RECV;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_IN,
                     &acnData);
    /* Remove UDP header. */
    MESSAGE_RemoveHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);

    /* Send packet to application layer. */
    MESSAGE_Send(node, msg, TRANSPORT_DELAY);
}
```

FIGURE 4-57. Processing a Packet Received from Network Layer at UDP

Function TransportUdpLayer calls function TransportUdpSendToNetwork when it receives a packet from the Application Layer. Function TransportUdpSendToNetwork, shown in Figure 4-58, performs the following actions:

1. Creates a header, using the API MESSAGE_Addheader.
2. Copies the source and destination port numbers from the message info field to the header.
3. Sets the length field of the header to the message packet size, using the API MESSAGE_ReturnPacketSize.
4. Sends the packet to the Network Layer, using the API NetworkIpReceivePacketFromTransportLayer.

   Function NetworkIpReceivePacketFromTransportLayer, defined in network_ip.cpp, is the API for sending a packet from the Transport Layer to the Network Layer. TransportUdpSendToNetwork passes to NetworkIpReceivePacketFromTransportLayer an integer parameter that is the IP Protocol Number for the Transport Layer protocol. In Figure 4-58, this parameter is `IPPROTO_UDP`, which is defined in EXATA_HOME/libraries/developer/src/network_ip.h (see Section 4.3.5.8).

```
void
TransportUdpSendToNetwork(Node *node, Message *msg)
{
    TransportDataUdp *udp = (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader *udpHdr;
    AppToUdpSend *info;

    if (udp->udpStatsEnabled == TRUE)
    {
        udp->statistics->numPktFromApp++;
    }
    MESSAGE_AddHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);
    udpHdr = (TransportUdpHeader *) msg->packet;
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);
    udpHdr->sourcePort = info->sourcePort;
    udpHdr->destPort = info->destPort;
    udpHdr->length = (unsigned short) MESSAGE_ReturnPacketSize(msg);
    udpHdr->checksum = 0;  /* checksum not calculated */

    ActionData acnData;
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_OUT,
                     &acnData);
    NetworkIpReceivePacketFromTransportLayer(
        node,
        msg,
        info->sourceAddr,
        info->destAddr,
        info->outgoingInterface,
        info->priority,
        IPPROTO_UDP,
        FALSE,
        info->ttl);
}
```

FIGURE 4-58. Processing a Packet Received from Application Layer at UDP

4.3.5.6.2.2 RSVP-TE Event Dispatcher

RSVP-TE is a control protocol operating at the Transport Layer. Unlike UDP and TCP, RSVP-TE is not used for transporting Application Layer data; therefore, it does not exchange packets with the Application Layer. Packets received from the Network Layer are processed at the Transport Layer and are not forwarded to the Application Layer.

See files transport_rsvp.h and transport_rsvp.cpp in EXATA_HOME/libraries/multimedia_enterprise/src for definitions of data structures and functions used for implementing RSVP-TE.

Function RsvpLayer, shown in Figure 4-59, is the event dispatcher for RSVP-TE. Event `MSG_TRANSPORT_RSVP_InitApp` is a trigger to initialize RSVP-TE, and function RsvpInitApplication is called to handle this event. Event `MSG_TRANSPORT_FromNetwork` corresponds to receiving a packet from the Network Layer and function RsvpHandlePacket is called to handle this event. All other events are timer events and are handled by their respective event handling routines.

The event dispatcher also includes a default case in the switch statement to handle events of an unknown type.

Note that after processing each event, RsvpLayer calls API MESSAGE_Free to free the message.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

```
void RsvpLayer(Node* node, Message* msg)
{
    switch (MESSAGE_GetEvent(msg))
    {
        case MSG_TRANSPORT_RSVP_InitApp:
        {
            RsvpInitApplication(node, msg);
            break;
        }

        case MSG_TRANSPORT_FromNetwork:
        {
            NetworkToTransportInfo* netToTransInfo =
                (NetworkToTransportInfo*) MESSAGE_ReturnInfo(msg);

            RsvpHandlePacket(node, msg,
                             GetIPv4Address(netToTransInfo->sourceAddr));
            break;
        }

        case MSG_TRANSPORT_RSVP_PathRefresh:
        {
           ...
        }

        case MSG_TRANSPORT_RSVP_ResvRefresh:
        {
            ...
        }

        case MSG_TRANSPORT_RSVP_HelloExtension:
        {
            ...
        }

        case MSG_TRANSPORT_RSVP_InitiateExplicitRoute:
        {
            ...
        }

        default:
        {
            // invalid event type
            ERROR_Assert(FALSE, "Invalid RSVP message.");
            break;
        }
    }
    // Free the message
    MESSAGE_Free(node, msg);
}
```

FIGURE 4-59. Event Dispatcher for RSVP-TE

4.3.5.7 Integrating with the Application Layer

Application Layer protocols exchange data by using transport services provided by UDP or TCP by means of APIs listed in Section 4.3.4.1 and Section 4.3.4.2. Since TCP is a connection-oriented protocol, it also implements connection management APIs, such as APP_TcpOpenConnection, APP_TcpServerListen and APP_TcpCloseConnection. These APIs are implemented in EXATA_HOME/main/app_util.cpp. As an example, function APP_UdpSendNewDataWithPriority, shown in Figure 3-7, sends application data to UDP with a user-specified priority after a user-specified delay. Section 3.3.2.1.2 explains the implementation of APP_UdpSendNewDataWithPriority.

If the new protocol, MYPROTOCOL, provides a transport service, integrate MYPROTOCOL with the Application Layer as follows:

1. Write functions similar to the UDP and TCP APIs which enable Application Layer protocols to use the services provided by MYPROTOCOL. Include these functions in the file transport_myprotocol.cpp.
2. Include the prototypes of these functions in the file transport_myprotocol.h.
3. Make these functions available to any specific Application Layer protocol by including the file transport_myprotocol.h in Application Layer protocol's source file, e.g., to enable the Application Layer protocol CBR to use MYPROTOCOL, include the following statement in the file EXATA_HOME/libraries/developer/src/app_cbr.cpp:

```
#include "transport_myprotocol.h"
```

4.3.5.8 Integrating with the Network Layer

When the IP protocol at the Network Layer sends a packet to a protocol at the Transport Layer protocol, it calls a function specific to that protocol. Therefore, when a new protocol, MYPROTOCOL, is added at the Transport Layer, a function needs to be added at the Network Layer which enables IP to deliver packets to MYPROTOCOL. This section describes the steps for integrating a new Transport Layer protocol with the Network Layer.

1. Define an IP Protocol Number for MYPROTOCOL. File network_ip.h contains constant definitions for all Transport Layer and Network Layer protocols (see Figure 4-60). For example, the IP Protocol Number for UDP is 17. Add a constant definition to associate an IP Protocol Number with MYPROTOCOL.

Note **Be sure to use an IP Protocol Number that is not already used for some other protocol.**

```
//--------------------------------------------------------
// IP protocol numbers
//--------------------------------------------------------

// IP protocol numbers for network-layer and transport-layer protocols

...
// /**
// CONSTANT    :: IPPROTO_TCP : 6
// DESCRIPTION :: IP protocol number for TCP.
// **/
#define IPPROTO_TCP             6
// /**
// CONSTANT    :: IPPROTO_UDP : 17
// DESCRIPTION :: IP protocol number for UDP.
// **/
#define IPPROTO_UDP             17
// /**
// CONSTANT    :: IPPROTO_RSVP : 46
// DESCRIPTION :: IP protocol number for RSVP.
// **/
#define IPPROTO_RSVP            46
...
// /**
// CONSTANT    :: IPPROTO_MYPROTOCOL : 255
// DESCRIPTION :: IP protocol number for MYPROTOCOL.
// **/
#define IPPROTO_MYPROTOCOL      255
...
```

FIGURE 4-60. Declaring IP Protocol Number for MYPROTOCOL

2. Modify function DeliverPacket in network_ip.cpp. Function DeliverPacket, shown in Figure 4-61, performs a switch on the IP Protocol Number, `ipProtocolNumber`, contained in the IP header of the received packet, and calls the appropriate routine to deliver the packet to the protocol identified by `ipProtocolNumber`. For example, DeliverPacket calls the function SendToUdp to deliver a packet to UDP if the IP Protocol Number read from the packet's header is 17.

Add code to DeliverPacket to call function SendToMyprotocol, with appropriate parameters, if the IP Protocol Number read from the packet's header is IPPROTO_MYPROTOCOL (defined in step 1), as shown in Figure 4-61.

```
static void
DeliverPacket(Node *node, Message *msg,
              int interfaceIndex, NodeAddress previousHopAddress)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    NodeAddress sourceAddress = 0;
    NodeAddress destinationAddress =0;
    unsigned char ipProtocolNumber;
    unsigned ttl =0;
    TosType priority;
    ...
    IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;
    ...
    ipProtocolNumber = ipHeader->ip_p;
    ...
    switch (ipProtocolNumber)
    {
        // Delivery to Transport Layer protocols.

        case IPPROTO_UDP:
        {
            SendToUdp(node, msg, priority, sourceAddress, destinationAddress,
                      interfaceIndex);
            break;
        }
        case IPPROTO_TCP:
        {
            SendToTcp(node, msg, priority, sourceAddress, destinationAddress,
                      aCongestionExperienced);
            break;
        }
        ...
        case IPPROTO_RSVP:
        {
            SendToRsvp(node, msg, priority, sourceAddress,
                       destinationAddress, interfaceIndex, ttl);
            break;
        }
        ...
        case IPPROTO_MYPROTOCOL:
        {
            SendToMyprotocol(node, msg, priority, sourceAddress,
                             destinationAddress, ...);
            break;
        }
        ...
    }
    ...
}
```

FIGURE 4-61. Delivering Packets from IP to Transport Layer Protocols

3. Write function SendToMyprotocol to deliver a packet from the Network Layer to MYPROTOCOL. Include this function in the file network_ip.cpp and specify its prototype in the file network_ip.h. Follow the example of function SendToUdp, which delivers a packet from the Network Layer to UDP. Function SendToUdp is shown in Figure 4-62 and is implemented in network_ip.cpp.

```
void
SendToUdp(
    Node *node,
    Message *msg,
    TosType priority,
    NodeAddress sourceAddress,
    NodeAddress destinationAddress,
    int incomingInterfaceIndex)
{
    NetworkToTransportInfo *infoPtr;

    MESSAGE_SetEvent(msg, MSG_TRANSPORT_FromNetwork);
    MESSAGE_SetLayer(msg, TRANSPORT_LAYER, TransportProtocol_UDP);
    MESSAGE_InfoAlloc(node, msg, sizeof(NetworkToTransportInfo));

    infoPtr = (NetworkToTransportInfo *) MESSAGE_ReturnInfo(msg);

    SetIPv4AddressInfo(&infoPtr->sourceAddr, sourceAddress);
    SetIPv4AddressInfo(&infoPtr->destinationAddr, destinationAddress);

    infoPtr->priority = priority;
    infoPtr->incomingInterfaceIndex = incomingInterfaceIndex;

    MESSAGE_Send(node, msg, PROCESS_IMMEDIATELY);
}
```

FIGURE 4-62. Sending Packets from IP to UDP

4.3.5.9 Collecting and Reporting Statistics

In this section, we describe how to collect and report statistics for a Transport Layer protocol.

4.3.5.9.1 Declaring Statistics Variables

A Transport Layer protocol can be configured to record statistics specified by the programmer, such as:

- Number of packets sent to the Application Layer
- Number of packets received from the Application Layer

To enable statistics collection for the protocol, include the statistic collection variables in the structure used to hold the protocol state (see Section 4.3.5.4). The statistics related variables can also be defined in a structure and then that structure is included in the state variable. For example, the data structure for UDP, `TransportDataUdp`, contains a pointer to the UDP statistics variable, `TransportUdpStat`, shown below:

```
typedef struct {
    D_Int32 numPktFromApp;
    D_Int32 numPktToApp;
} TransportUdpStat;
```

`TransportDataUdp` and `TransportUdpStat` are defined in transport_udp.h.

Also, include a variable in the protocol state structure that indicates whether statistics collection is enabled for the protocol. For example, field `udpStatsEnabled` of `TransportDataUdp` is a Boolean flag that indicates whether statistics collection is enabled for UDP.

4.3.5.9.2 Initializing Statistics

Initialize statistics variables in the protocol's initialization function. Determine whether statistics collection is enabled for the protocol and set the statistics collection flag accordingly. For example, field `udpStatsEnabled` of `TransportDataUdp` is set to TRUE or FALSE by the initialization function TransportUdpInit depending upon the input configuration, as shown in Figure 4-53. Function TransportUdpInit allocates memory for the statistics variable `TransportUdpStat` and initializes all fields of `TransportUdpStat` to 0, if UDP statistics collection is enabled.

4.3.5.9.3 Updating Statistics

After declaring and initializing the statistics variables, update their value during the protocol life cycle, as required. For example, UDP increments the value of `numPktToApp` in function TransportUdpsendToApp (see Figure 4-57) every time UDP sends a packet to the Application Layer, as shown below:

```
...
 if (udpLayer->udpStatsEnabled == TRUE)
 {
     udpLayer->statistics->numPktToApp++;
 }
...
```

4.3.5.9.4 Printing Statistics

As a final step towards statistics collection, create a function to print statistics. Call this function from the finalization function of the protocol, which is discussed in Section 4.3.5.9.5. Alternatively, statistics can be printed directly in the finalization function, as shown in Figure 4-64.

4.3.5.9.5 Adding Dynamic Statistics

Dynamic statistics are statistic variables whose values can be observed in the EXata GUI during the simulation. See Section 5.2.3 for adding dynamic statistics to a protocol. Refer to *EXata User's Guide* for details of viewing dynamic statistics during the simulation.

4.3.5.10 Finalization

The finalization function of the protocol is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

At the end of simulation, the finalization function for each protocol is called to print the protocol statistics. As discussed in Section 3.4.3, the finalization function is called hierarchically. The node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp, calls the finalization function for Transport Layer, TRANSPORT_Finalize, defined in transport.cpp. TRANSPORT_Finalize calls the finalization function of each Transport Layer protocol running at the node.

4.3.5.10.1 Modifying the Transport Layer Finalization Function

Call the finalization function of MYPROTOCOL, MyprotocolFinalize, from the Transport Layer finalization function, TRANSPORT_Finalize, defined in transport.cpp. If the protocol is a mandatory protocol like UDP and TCP, then make a call to the protocol's finalization function unconditionally. If the protocol is a non-mandatory protocol like RSVP-TE, then check if the protocol is enabled before making a call to the protocol's finalization function. Figure 4-63 shows the outline of code that needs to be added to TRANSPORT_Finalize.

```
void
TRANSPORT_Finalize(Node * node)
{
    TransportUdpFinalize(node);
    TransportTcpFinalize(node);
    ...
    if (node->transportData.myprotocolEnabled == TRUE)
    {
        MyprotocolFinalize(node);
    }
}
```

FIGURE 4-63. Transport Layer Finalization Function

4.3.5.10.2 Implementing the Protocol Finalization Function

Write the finalization function for protocol, MyprotocolFinalize. If statistics collection is enabled for MYPROTOCOL, call a function to print the protocol statistics (see Section 4.3.5.9.4), or add code directly to MyprotocolFinalize to print statistics. UDP follows the latter approach. Function TransportUdpFinalize, shown in Section 4-64 and implemented in transport_udp.cpp, is the finalization function for UDP. Use TransportUdpFinalize as a template to write MyprotocolFinalize.

Function TransportUdpFinalize calls the C function sprintf to create a single string containing the statistic name and statistic value, and then calls function IO_PrintStat to print that string to a file. Function IO_PrintStat function, defined in EXATA_HOME/include/fileio.h, requires the following parameters:

- Node pointer: Pointer to the node reporting the statistics.
- Layer: String indicating the layer. Set this to `"Transport"` for the Transport Layer.
- Protocol: String indicating the protocol name.
- Interface address: Interface address. Set this to `ANY_DEST` for Transport Layer protocols.
- Instance identifier: Instance identifier or port number. Set this to -1 if there is no instance identifier.
- Buffer: String containing the statistics.

```
void
TransportUdpFinalize(Node *node)
{
    char buf[MAX_STRING_LENGTH];
    TransportDataUdp* udp = node->transportData.udp;
    TransportUdpStat* st =  udp->statistics;


    if (udp->udpStatsEnabled == TRUE) {

        sprintf (buf, "Packets from Application Layer = %d",
                      st->numPktFromApp);
        IO_PrintStat(
            node,
            "Transport",
            "UDP",
            ANY_DEST,
            -1 /* instance Id */,
            buf);

        sprintf (buf, "Packets to Application Layer = %d",
                      st->numPktToApp);
        IO_PrintStat(
            node,
            "Transport",
            "UDP",
            ANY_DEST,
            -1 /* instance Id */,
            buf);
    }
}
```

FIGURE 4-64. Finalization Function for UDP

Like all other functions, specify the prototype of the finalization function, MyprotocolFinalize, in the protocol's header file, transport_myprotocol.h.

4.3.5.11 Including and Compiling Files

The final step in integrating your transport model into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the transport model in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Developer library, then modify EXATA_HOME/libraries/developer/Makefile-common as shown in Figure 4-65. Recompile EXata after making the changes.

```
...
# common sources
#
DEVELOPER_SRCS = \
$(DEVELOPER_SRCDIR)/adaptation_aal5.cpp \
$(DEVELOPER_SRCDIR)/adaptation.cpp \
...
$(DEVELOPER_SRCDIR)/transport_abstract_tcp_timer.cpp \
$(DEVELOPER_SRCDIR)/transport_abstract_tcp_usrreq.cpp \
$(DEVELOPER_SRCDIR)/transport_in_pcb.cpp \
$(DEVELOPER_SRCDIR)/transport_myprotocol.cpp \
$(DEVELOPER_SRCDIR)/transport_tcp.cpp \
$(DEVELOPER_SRCDIR)/transport_tcp_input.cpp \
$(DEVELOPER_SRCDIR)/transport_tcp_output.cpp \
...
```

FIGURE 4-65. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.3.5.12 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.3.6 Special Issues for Transport Layer Protocols

4.3.6.1 Setting Address for Broadcast Messages

For broadcasting packets, the destination address can be set to ANY_DEST, or the library function NetworkIpGetInterfaceBroadcastAddress can be used to get the interface broadcast address. ANY_DEST is a constant defined in EXATA_HOME/include/main.h and stands for any destination. Function NetworkIpGetInterfaceBroadcastAddress is defined in network_ip.h, and returns the broadcast address of the specified interface.

Figure 4-66 shows a code snippet from the Bellmanford function SendRouteAdvertisement (implemented in file EXATA_HOME/libraries/developer/src/routing_bellmanford.cpp) that sets the destination address to the interface broadcast address for the wired interface and to ANY_DEST for the wireless interface.

```
static void SendRouteAdvertisement(
             Node *node, RouteAdvertisementType type)
 {
  ...
  int i;
  for (i = 0; i < node->numberInterfaces; i++)
    {
      NodeAddress destAddress;
      ...
      if (NetworkIpIsWiredNetwork (node, i))
       {
         destAddress =
                 NetworkIpGetInterfaceBroadcastAddress(node, i);
       }
    else
     {
       destAddress = ANY_DEST;
     }
   ...
   }
  ...
}
```

FIGURE 4-66. Setting Broadcast Address

4.4 Network Layer

The Network Layer resides between the Transport and MAC Layers in the EXata protocol stack, as shown in Figure 4-1. The Network Layer provides a communication service to support the Transport Layer between two processes in two different hosts. In particular, the Network Layer moves the Transport Layer data from the source host to the destination host. It is the lowest layer to deal with the end-to-end issue. The main issues related to this layer are:

- Routing: Determining the next hop and outgoing interface for a packet
- Traffic Control: Controlling congestion and transmission rate
- Addressing: Identifying nodes in the network
- Internetworking: Interconnecting heterogeneous networks

This section gives a detailed description of how to add a Network Layer protocol to EXata.

4.4.1 Network Layer Protocols in EXata

EXata provides an implementation of a large number of Network Layer protocols, which can be grouped into the categories listed below.

- Network protocols
- Unicast routing protocols
- Multicast routing protocols
- Queues
- Schedulers

4.4.1.1 Network Protocols

Table 4-7 lists the network protocols in EXata.

TABLE 4-7. Network Protocols in EXata

Model/Protocol	Description	Model Library
Fixed Comms	Fixed Communications Model. The Fixed Communications model enables the user to enforce a minimum drop rate for application packets and, optionally, a fixed delay for packets that are not dropped.	Developer
ICMP	Interet Control Message Protocol. ICMP is an integral part of IPv4. It allows a router or destination host to communicate with the source, to report an error in IP datagram processing and for diagnostic or routing purposes.	Developer
ICMPv6	Interet Control Message Protocol version6. ICMPv6 is an integral part of IPv6. It is used by IPv6 nodes to report errors encountered in processing packets, and to perform other internet-layer functions, such as diagnostics (ICMPv6 "ping").	Developer

TABLE 4-7. Network Protocols in EXata (Continued)

Model/Protocol	Description	Model Library
IGMP	Internet Group Management Protocol. IGMP is a multicast group management protocol. It is used between routers and hosts involved in multicast data traffic. It operates between a host sending or receiving multicast data and its directly attached router.	Developer
Dual IP	Dual IP. The Dual IP protocol provides a means to IPv6 routers and hosts to maintain compatibility with IPv4 hosts and routers by enabling IPv6 routers/hosts to carry an IPv4/IPv6 datagram over the attached IPv4 infrastructures by encapsulating the datagram within an IPv4 datagram.	Developer
IPv4	Internet Protocol version 4. The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. The Internet Protocol provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. The Internet Protocol also provides for fragmentation and reassembly of long datagrams.	Developer
IPv6	Internet Protocol version 6. IPv6 is the newer version of the Internet Protocol that is meant to replace IPv4. IPv6 provides for an increased address space (from 32 bits to 128 bits). Similar to IPv4, IPv6 also provides functionalities such as forwarding, fragmentation, automatic discovery of default routers, fastest forwarding process using destination cache, neighbor cache and prefix list, hierarchical networking, and simplified packet format.	Developer
Mobile IPv4	Mobile Support for IPv4. Mobility IPv4 defines a protocol that allows transparent routing of IP datagrams to Mobile Nodes as they move about from one domain to another on the Internet.	Multimdia and Enterprise
NDP	Neighbor Discovery Protocol. NDP is used by nodes (hosts and routers) in an IPv6 network to determine the link-layer addresses for neighbors known to reside on attached links, to find neighboring routers that are willing to forward packets on their behalf, to actively keep track of which neighbors are reachable and which are not, and to detect changed link-layer addresses.	Developer

4.4.1.2 Routing Protocols

In general, routing refers to moving information across an internetwork and involves the following two activities:

- Determination of an optimal path from a source to a destination
- Transporting packets through an internetwork (also referred to as switching)

A routing algorithm determines an optimal path and stores the information in routing tables. Routers communicate with one another and maintain their routing tables through the transmission of a variety of

messages. Based on the information stored in the routing table of a router, a packet is forwarded to the next node on the path to the destination.

EXata provides a variety of routing protocols, which can be grouped into the following categories:

- Unicast routing for wireless ad hoc networks
- Unicast routing for wired networks
- Unicast routing for mixed networks
- Multicast routing for wireless networks
- Multicast routing for wired networks

Unicast Routing

Routing can be done in a variety of ways. Depending upon the underlying network technology and topology, different choices can be made. In general, routing protocols can be viewed as being either proactive or reactive. While a proactive routing strategy is suitable for wired networks, it may not be a good choice for mobile ad hoc networks. On the other hand, reactive routing strategies generally work well for ad hoc networks but can cause extra overhead for wired networks.

- **Wireless Ad Hoc Networks**

  One of the primary characteristics of an ad hoc network is that the network topology is constantly changing. For this reason, it is recommended that routes be evaluated only on an as-needed basis.

  Reactive routing protocols normally have two types of routing packets: *request*/*discovery* and *reply* packets. Request packets are normally flooded while reply packets are unicast. These types of routing protocols,e.g., AODV, DSR and LAR1, are efficient if routes are used infrequently and may be sub-optimal.

  Some proactive routing protocols are also suitable for ad hoc networks. An example of this type of routing protocol is OLSR. Proactive routing protocols mainly use two types of broadcasts: periodic and triggered. They are suitable if routes are used frequently and routes always need to be optimal.

- **Wired Networks**

  Wired networks are comparatively stable and topology changes occur less frequently than in wireless ad hoc networks.

- **Mixed Networks**

  AODV, DYMO. OSPFv2 and OSPFv3 are some of the Network Layer routing protocols that can be used throughout a mixed network, such as switched ethernet, point-to-point, and wireless ad hoc networks connected together.

Table 4-8 lists the unicast routing protocols (implemented at the Network Layer) in EXata. The table also specifies whether a routing protocol is a proactive or on-demand protocol, and if it is supported in IPv4 networks, IPv6 networks, or both. See the corresponding model library for a detailed description of each protocol and its parameters.

Some routing protocols are implemented at the Application Layer (see Table 4-3)..

TABLE 4-8. Unicast Routing Protocols in EXata

Unicast Routing Protocol	Description	Type	IP Version(s)	Model Library
AODV	Ad-hoc On-demand Distance Vector (AODV) routing protocol.	On-demand	IPv4, IPv6	Wireless
DSR	Dynamic Source Routing (DSR) protocol.	On-demand	IPv4	Wireless
DYMO	DYnamic MANET On-demand (DYMO) routing protocol.	On-demand	IPv4, IPv6	Wireless

TABLE 4-8. Unicast Routing Protocols in EXata (Continued)

Unicast Routing Protocol	Description	Type	IP Version(s)	Model Library
FSRL	Landmark Ad-hoc Routing (LANMAR) protocol. This protocol uses Fisheye as the local scope routing protocol.	Proactive	IPv4	Wireless
IARP	IntrA-zone Routing Protocol (IARP). This is a vector-based proactive routing protocol and is a component of ZRP.	Proactive	IPv4	Wireless
IERP	Inter-zone Routing Protocol (IERP). This is an on-demand routing protocol and is a component of ZRP.	On-demand	IPv4	Wireless
LAR1	Location-Aided Routing (LAR) protocol, version 1. This protocol utilizes location information to improve scalability of routing.	On-demand	IPv4	Wireless
OSPFv2	Open Shortest Path First (OSPF) routing protocol, version 2. This is a link state-based routing protocol for IPv4 networks.	Proactive	IPv4	Multimedia and Enterprise
OSPFv3	Open Shortest Path First (OSPF) routing protocol, version 3. This is a link state-based routing protocol for IPv6 networks.	Proactive	IPv6	Multimedia and Enterprise
STAR	Source Tree Adaptive Routing (STAR) protocol.	Proactive	IPv4	Wireless
ZRP	Zone Routing Protocol (ZRP). ZRP uses IARP for intrazone routing and IERP for interzone routing. It uses BRP to optimiz routing between perimeter nodes.	Hybrid (Proactive and On-demand)	IPv4	Wireless

Multicast Routing

EXata implements several multicast routing protocols for wired and wireless networks.

- **Wireless Networks**

  Although other multicast routing protocols can be used for wireless networks, ODMRP is the recommended routing protocol for wireless networks. ODMRP uses a mesh-based multicast scheme and a forwarding group concept. The on-demand routing technique used by ODMRP helps to reduce channel overhead and to improve scalability.

- **Wired Networks**

  Wired multicast routing protocols build a source-based multicast delivery tree. The tree is built when a router receives the first multicast data packet for a particular group. All of these protocols have their own forwarding function that is called when a node needs to forward a multicast data packet.

Table 4-9 lists the different multicast routing protocols in EXata. The table also specifies whether a routing protocol is a proactive or on-demand protocol. See the corresponding model library for a detailed description of each protocol and its parameters.

Note **All multicast protocols listed in Table 4-9 are supported only for IPv4 networks.**

TABLE 4-9. Multicast Routing Protocols in EXata

Multicast Routing Protocol	Description	Type	Model LIbrary
DVMRP	Distance Vector Multiple Routing Protocol (DVMRP) DVMRP is a multicast routing protocol designed for wired networks. It is a tree-based multicast scheme that uses reverse path multicasting.	Proactive	Multimedia and Enterprise
MOSPF	Multicast Open Path Shortest First (MOSPF) protocol. This is a multicast extension of OSPFv2. MOSPF is a pruned tree-based, multicast scheme that takes advantage of commonality of paths from source to destinations.	Proactive	Multimedia and Enterprise
ODMRP	On-Demand Multicast Routing Protocol (ODMRP). This is a mesh-based, wireless ad-hoc routing protocol for single subnets. It applies a soft on-demand procedures to build routes and uses soft state to maintain multicast group membership.	On-demand	Wireless
PIM	Protocol Independent Multicast (PIM) routing protocol. PIM relies on an underlying topology-gathering protocol to populate a routing table with routes. The routing table provides the next hop router along a multicast-capable path to each destination subnet. Both sparse mode and dense mode versions of the protocol are supported.	Proactive	Multimedia and Enterprise

4.4.1.3 Queues

Table 4-10 lists the different queue models in EXata. See the corresponding model library for a detailed description of each model and its parameters.

TABLE 4-10. Queue Models in EXata

Queue	Description	Model Library
FIFO	First In First Out (FIFO) queue. This is the basic queue type and is also called the Drop Tail queue. Packets are enqueued as long as there is buffer space available. If the queue is full when a packet arrives, the packet is dropped.	Developer
RED	Random Early Drop (RED) queue. This queue is similar to FIFO, except that when the queue length exceeds a certain threshold, arriving packets are randomly dropped with a probability that depends on the queue length.	Developer
RIO	RED with In/Out bit (RIO) queue. RIO is a multiple average multiple threshold variant of RED that operates two-color and three-color modes. Twin and three RED algorithms are used in two-color and three-color modes, respectively.	Developer
WRED	Weighted Random Early Drop (WRED) queue. WRED is a variant of RED and uses three RED algorithms for three drop precedence levels.	Developer

4.4.1.4 Schedulers

Table 4-11 lists the different scheduler models in EXata. See the corresponding model library for a detailed description of each model and its parameters.

TABLE 4-11. Scheduler Models in EXata

Scheduler	Description	Model Library
CBQ	Class-based queuing algorithm. This algorithm is usually used by DiffServ. Queues are divided into classes. The network protocol allocates bandwidth for each queue. Scheduling is based on the bandwidth available to each class.	Developer
DIFFSERV	Differentiated services (DiffServ) quality of service protocol. When this option is selected, DiffServ queues and schedulers are configured. The DiffServ scheduler for IP is a combination of two schedulers: the inner and outer schedulers. Generally, the weighted fair or weighted round-robin scheduler is chosen as the inner scheduler and the strict priority scheduler is chosen as the outer scheduler.	Multimedia and Enterprise
ROUND-ROBIN	Round-robin scheduler. Queues are scheduled in a round-robin fashion.	Developer
SELF-CLOCKED-FAIR	Self-clocked fair scheduler. Scheduling is based on the Self-Clocked Fair Queuing (SFFQ) algorithm.	Developer
STRICT-PRIORITY	Strict priority scheduler. Packets are scheduled strictly based on their priority. A packet is scheduled only when all higher priority queues are empty.	Developer
WEIGHTED-FAIR	Weighted fair scheduler. Scheduling is based on the Weighted Fair Queuing (WFQ) algorithm.	Developer
WEIGHTED-ROUND-ROBIN	Weighted Round-Robin (WRR) scheduler. This is a variant of the round-robin scheduler. The round-robin scheduler services one packet from each queue in turn. The WRR scheduler services multiple packets from each queue in turn, where the number of packets serviced depends on the queue's weight.	Developer

4.4.2 Network Layer Organization: Files and Folders

This section briefly examines the files and folders that are relevant to Network Layer protocols. These files contain detailed comments on functions and other code components.

The Network Layer API is composed of several macros, functions, and structures. These are defined in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/network.h

  This file contains definitions common to Network Layer protocols, the network data structure in the node structure, and the prototypes of functions defined in EXATA_HOME/main/network.cpp.

- EXATA_HOME/libraries/developer/src/network_ip.h

  This file contains definitions of data structure and parameters used in the IP implementation and prototypes of the functions defined in EXATA_HOME/libraries/developer/src/network_ip.cpp.
- EXATA_HOME/include/mapping.h

  This file contains definitions of data structures and functions used for determining the network type network protocols running at a node.
- EXATA_HOME/include/mac.h

  This file contains definitions of API functions needed to communicate with the MAC Layer.
- EXATA_HOME/include/if_queue.h

  This file contains definition of the class that implements queues and prototypes of functions related to queues.
- EXATA_HOME/include/if_scheduler.h

  This file contains definition of the class that implements schedulers and prototypes of functions related to schedulers.

The following header files are also relevant to the Network Layer:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.
- EXATA_HOME/include/buffer.h

  This file contains data structures and prototypes of functions for buffer operations.
- EXATA_HOME/include/list.h

  This file defines a generic doubly link list structure and prototypes of functions for list operations.

The following are the folders and source files associated with the Network Layer:

- EXATA_HOME/libraries/developer/src, EXATA_HOME/libraries/wireless/src

  This folder contains most of the Network Layer protocols implemented in EXata. These include network protocols, routing protocols, queuing protocols, and schedulers. The file names are indicative of the protocol for which they provide an implementation. Other libraries may contain code for Network Layer protocols as well.
- EXATA_HOME/main/network.cpp

  This file contains Network Layer functions, including the initialization, message processing, and finalization functions.
- EXATA_HOME/libraries/developer/src/network_ip.cpp

  This file contains functions that implement the IP protocol.

4.4.3 Network Layer Data Structures

The Network Layer data structures are defined in network.h and network_ip.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to files network.h and network_ip.h for a complete description.)

1. `NetworkRoutingProtocolType`**:** This enumeration type, defined in network.h, lists all the Network Layer protocols and all routing protocols, including those running at the Application Layer.

```
typedef enum
{
    NETWORK_PROTOCOL_IP = 0,
    NETWORK_PROTOCOL_IPV6,
    NETWORK_PROTOCOL_MOBILE_IP,
    ...
    ROUTING_PROTOCOL_AODV6,
    ROUTING_PROTOCOL_DYMO,
    ROUTING_PROTOCOL_DYMO6,
    ...
    ROUTING_PROTOCOL_NONE // this must be the last one
} NetworkRoutingProtocolType;
```

2. `NetworkRoutingAdminDistanceType`: This enumeration type, defined in network.h, assigns an administrative distance to each routing protocol that is included in the list. The administrative distance of a routing protocol determines its priority relative to other routing protocols when a route to a destination can be determined by more than one routing protocol. A protocol with a lower administrative distance has a higher priority.

```
typedef enum
{
    ROUTING_ADMIN_DISTANCE_STATIC = 1,
    ROUTING_ADMIN_DISTANCE_EBGPv4 = 20,
    ...
    ROUTING_ADMIN_DISTANCE_OLSR,
    ROUTING_ADMIN_DISTANCE_EIGRP,
    //StartRIP
    ROUTING_ADMIN_DISTANCE_RIP,
    //EndRIP
    ...

    // Should always have the highest administrative distance
    // (i.e., least important).
    ROUTING_ADMIN_DISTANCE_DEFAULT = 255
} NetworkRoutingAdminDistanceType;
```

3. `NetworkProtocolType`: This enumeration type, defined in EXATA_HOME/include/mapping.h, lists all network protocols supported in EXata.

```
typedef enum
{
    INVALID_NETWORK_TYPE,
    IPV4_ONLY,
    IPV6_ONLY,
    DUAL_IP,
    ATM_NODE,
    GSM_LAYER3,
    CELLULAR,
    NETWORK_VIRTUAL
}NetworkProtocolType;
```

4. `IpInterfaceInfoType`: This data structure, defined in network_ip.h, stores information for a specific interface, such as the routing protocol and scheduler running at the interface.

```
struct IpInterfaceInfoType
{
    // Constructor.  All member variables MUST be initialized in the
    // constructor.
    IpInterfaceInfoType();
    Scheduler* scheduler;
    Scheduler* inputScheduler;
    ...
    D_NodeAddress ipAddress;
    ...
    NetworkRoutingProtocolType              routingProtocolType;
    void*                                   routingProtocol;
    BOOL multicastEnabled;
    ...
    NetworkRoutingProtocolType multicastProtocolType;
    void *multicastRoutingProtocol;
    ...
    MacLayerAckHandlerType                  macAckHandler;
    ...
    // IPv6 interface information
    NetworkType interfaceType;
    BOOL isVirtualInterface;
    ...
    struct ipv6_interface_struct* ipv6InterfaceInfo;
    ...
};
```

5. `NetworkForwardingTableRow`: This data structure, defined in network_ip.h, stores one row of the forwarding table.

```
typedef struct
{
    NodeAddress destAddress;       // destination address
    NodeAddress destAddressMask;   // subnet destination Mask
    int interfaceIndex;            // index of outgoing interface
    NodeAddress nextHopAddress;    // next hop IP address
    int cost;
    // routing protocol type
    NetworkRoutingProtocolType protocolType;
    // administrative distance for the routing protocol
    NetworkRoutingAdminDistanceType adminDistance;
    BOOL interfaceIsEnabled;
} NetworkForwardingTableRow;
```

6. `NetworkForwardingTable`: This data structure, defined in network_ip.h, stores the forwarding table.

```
typedef struct
{
    int size;                         // Number of entries
    int allocatedSize;
    NetworkForwardingTableRow *row;  // Pointer to first row
} NetworkForwardingTable;
```

7. `NetworkMulticastForwardingTableRow`: This data structure, defined in network_ip.h, stores one row of the multicast forwarding table.

```
typedef struct
{
    NodeAddress sourceAddress;
    NodeAddress sourceAddressMask;        // Not used
    NodeAddress multicastGroupAddress;
    LinkedList *outInterfaceList;
} NetworkMulticastForwardingTableRow;
```

8. `NetworkMulticastForwardingTable`: This data structure, defined in network_ip.h, stores the multicast forwarding table.

```
typedef struct
{
    int size;                                    // Number of entries
    int allocatedSize;
    NetworkMulticastForwardingTableRow *row;  // Pointer to first row
} NetworkMulticastForwardingTable;
```

9. `NetworkDataIp`: This data structure, defined in network_ip.h, is the main data structure for the Network Layer and stores information about all Network Layer protocols running at the node.

```
typedef struct struct_network_ip_str
{
    ...
    NetworkForwardingTable      forwardTable;
    ...
    IpInterfaceInfoType* interfaceInfo[MAX_NUM_INTERFACES];
    ...
    NetworkIpStatsType stats;
    LinkedList                          *multicastGroupList;
    NetworkMulticastForwardingTable     multicastForwardingTable;
    ...
} NetworkDataIp;
```

10. `NetworkData`: This data structure stores the network protocol type running at the node and a pointer to the network protocol data structure.

```
struct struct_network_str
{
    NetworkDataIp* networkVar;  // IP state
    NetworkProtocolType networkProtocol;
    ...
    BOOL networkStats;  // TRUE if network statistics are collected
    //It is true if ARP is enabled
    BOOL isArpEnable;
    //It is true if RARP is enabled
    BOOL isRarpEnable;
    struct address_resolution_module *arModule;
    ...
};
```

4.4.4 Network Layer APIs and Inter-layer Communication

This section describes the API used by the Transport Layer to communicate with the Network Layer (see Section 4.4.4.1), the APIs used by the Network Layer to communicate with the Transport Layer (see Section 4.4.4.2), the APIs used by the Network Layer protocols to communicate with the MAC Layer (see Section 4.4.4.3), and the APIs used by the MAC Layer to communicate with the Network Layer (see Section 4.4.4.4). This section also lists some of the Network Layer utility APIs (see Section 4.4.4.5).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.4.4.1 Transport Layer to Network Layer Communication

The Transport Layer communicates with the Network Layer by using the API NetworkIpReceivePacketFromTransportLayer. This function sends a packet from a Transport Layer protocol (UDP, TCP or RSVP-TE) to the IP protocol at the Network Layer. The prototype for this function is contained in the file network_ip.h. The file network_ip.cpp contains the implementation of NetworkIpReceivePacketFromTransportLayer.

4.4.4.2 Network Layer to Transport Layer Communication

Several APIs are available for the Network Layer to communicate with the Transport Layer. The prototypes for these functions are contained in the file network_ip.h. The file network_ip.cpp contains the implementation of these functions.

Some of the APIs used for communication from the Network Layer to the Transport Layer are listed below.

- SendToUdp: This function sends a packet to the UDP protocol at the Transport Layer.
- SendToTcp: This function sends a packet to the TCP protocol at the Transport Layer.
- SendToRsvp: This function sends a packet to the RSVP-TE protocol at the Transport Layer.

4.4.4.3 Network Layer to MAC Layer Communication

A number of APIs are available at the Network Layer to communicate with the MAC Layer. The prototypes for the API functions are contained in the files network_ip.h and EXATA_HOME/include/mac.h. The files network_ip.cpp and EXATA_HOME/main/mac.cpp contain the implementation of these functions.

Some of the APIs used for communication from the Network Layer to the MAC Layer are listed below.

- NetworkIpSendRawMessage: This function adds an IP header to a packet and calls function RoutePacketAndSendToMac to add routing information to the packet.
- NetworkIpSendRawMessageWithDelay: This function adds an IP header to a packet and calls function RoutePacketAndSendToMac, after a specified delay, to add routing information to the packet.
- NetworkIpSendRawMessageToMacLayer: This function adds an IP header to a packet and sends the packet to the MAC Layer.
- NetworkIpSendRawMessageToMacLayerWithDelay: This function adds an IP header to a packet and sends the packet to the MAC Layer after a specified delay.
- NetworkIpSendPacketToMacLayer: This function sends an IP packet to the MAC Layer.
- NetworkIpSendPacketToMacLayerWithNewStrictSourceRoute: This function appends a new source route to an IP packet and sends the packet to the MAC Layer.
- MAC_NetworkLayerHasPacketToSend: This function is used by the Network Layer to inform the MAC Layer that a packet is ready to be sent.

4.4.4.4 MAC Layer to Network Layer Communication

MAC Layer protocols use several APIs to communicate with the Network Layer (see Section 4.5.4.2). These APIs, in turn, call functions implemented at the Network Layer. The prototypes for these Network Layer functions are contained in the file network_ip.h or network.h, and the implementation of these functions are contained in the file network_ip.cpp or EXATA_HOME/main/network.cpp.

Some of the Network Layer functions used for communication from the MAC Layer to the Network Layer are listed below.

- NetworkIpOutputQueueIsEmpty: This functions checks if the output queue at an interface is empty.
- NetworkIpOutputQueueDequeuePacket: This function dequeues a packet from an output queue.
- NetworkIpOutputQueueTopPacket: This function is used to view the top packet of a queue without dequeuing it.
- NetworkIpOutputQueueDequeuePacketForAPriority: This function dequeues a specific priority packet from an output queue.
- NETWORK_ReceivePacketFromMacLayer: This function is used by the MAC Layer to pass an incoming packet to the Network Layer.
- NetworkIpReceiveMacAck: This function notifies the Network Layer that a packet has been successfully delivered by the MAC protocol.

- NetworkIpNotificationOfPacketDrop: This function notifies the upper layer protocols when a packet is dropped at the MAC Layer.

4.4.4.5 Network Layer Utility APIs

Several APIs are available at the Network Layer that perform tasks internal to the Network Layer. Some of these functions can be used by other layers as well. The prototypes for these API functions are contained in the file network_ip.h. The file network_ip.cpp contains the implementation of these functions.

Some of the Network Layer utility APIs are listed below.

- NetworkIpSetPromiscuousMessagePeekFunction: This function registers the function that promiscuously peeks at packets with IP.
- NetworkIpSetMacLayerAckHandler: This function registers the function that processes MAC Layer acknowledgements with IP.
- NetworkIpSetRouterFunction: This function registers a routing protocol's router function with IP.
- NetworkIpGetRouterFunction: This function is used by IP to get a pointer to the router function used by a routing protocol at a given interface.
- NetworkIpGetInterfaceAddress: This function returns the node address for the specified interface.

4.4.5 Adding a Network Layer Unicast Routing Protocol

This section provides an overview of the flow of a Network Layer unicast routing protocol and provides an outline for developing and adding a new Network Layer unicast routing protocol to EXata. It describes how to develop code components common to most routing protocols such as initializing, sending and receiving packets, determining routes, and collecting statistics.

We illustrate the process of adding a Network Layer unicast routing protocol by using as an example the implementation code for the AODV (Ad Hoc On-demand Distance Vector) routing protocol. The header file for the AODV implementation is routing_aodv.h and the source file is routing_aodv.cpp in the folder EXATA_HOME/libraries/wireless/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a Network Layer unicast routing protocol. After understanding the discussed snippets, look at the complete code for AODV to understand how a Network Layer unicast routing protocol is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a Network Layer routing protocol to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.4.5.2).
2. Modify the file network_ip.cpp to include the protocol's header file (see Section 4.4.5.2).
3. Include the protocol in the list of Network Layer protocols and trace protocols (see Section 4.4.5.3).
4. Define data structures for the protocol (see Section 4.4.5.4).
5. Decide on the format for the protocol-specific configuration parameters (see Section 4.4.5.5.1).
6. Call the protocol's initialization function from the routing initialization function, IpRoutingInit (see Section 4.4.5.5.2).
7. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Read and store the configuration parameters (see Section 4.4.5.5.3.1).
   b. Initialize the state variables and routing table (see Section 4.4.5.5.3.2).
   c. Register the protocol's callback functions with IP (see Section 4.4.5.5.3.3).

   d. Initialize timers (see Section 4.4.5.5.3.4).

8. Call the protocol event dispatcher from the IP event dispatcher, NetworkIpLayer (see Section 4.4.5.6.1).
9. Declare any new event types used by the protocol in the header file EXATA_HOME/include/api.h (see Section 4.4.5.6.2).
10. Write the protocol event dispatcher (see Section 4.4.5.6.2).
11. Modify the IP function NetworkRoutingGetAdminDistance (see Section 4.4.5.7).
12. Implement the protocol's routing packet handler.
    a. Define an IP Protocol Number for the protocol (see Section 4.4.5.8.1).
    b. Write a function to handle routing packets (see Section 4.4.5.8.2).
    c. Call the routing packet handler function from the IP function DeliverPacket (see Section 4.4.5.8.1).
13. Write the router function and any other call back functions used by the protocol (see Section 4.4.5.9).
14. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.4.5.10.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.4.5.10.2).
    c. Update the statistics as appropriate (see Section 4.4.5.10.3).
    d. Write a function to print the statistics (see Section 4.4.5.10.4).
    e. Add dynamic statistics to the protocol, if desired (see Section 4.4.5.10.5) .
15. Call the protocol finalization function from the IP finalization function, NetworkIpFinalize (see Section 4.4.5.11.1).
16. Write the protocol finalization function (see Section 4.4.5.11.2). Call the function to print statistics from the protocol finalization function.
17. Include the protocol header and source files in the EXata tree and compile (see Section 4.4.5.12).
18. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.4.5.13).

4.4.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new protocol, the programmer name the different components of the new protocol in a similar manner. It will be helpful to examine the implementation of AODV in EXata for hints for naming and coding different components of the new protocol.

In this section, we describe the steps for developing a Network Layer unicast routing protocol called "MYPROTOCOL". We will use the string "Myprotocol" in the names of the different components of this protocol, just as the string "Aodv" appears in the names of the components of the AODV implementation.

4.4.5.2 Creating Files

The first step towards adding a Network Layer routing protocol is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/developer/src. However, it is recommended that all user-developed models be made part of a library. In our example, we will place the routing protocol in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn't already exist, create a directory in EXATA_HOME/library called user_models and a subdirectory in EXATA_HOME/library/user_models called src. Create the files for the routing protocol and place them in the folder EXATA_HOME/library/user_models/src. Name these files in a way that clearly

indicates the model that they implement. For unicast routing protocols, prefix the file names with *routing_*. For multicast routing protocols, prefix the file names with *multicast_*.

Examples:

- routing_aodv.cpp, routing_aodv.h: Implement AODV (Ad Hoc On-demand Distance Vector routing protocol)
- routing_dsr.cpp, routing_dsr.h: Implement DSR (Dynamic Source Routing protocol)
- multicast_mospf.cpp, multicast_mospf.h: Implement the MOSPF (Multicast Open Shortest Path First) protocol.

In keeping with the naming guidelines of Section 4.4.5.1, the header file for the example protocol is called routing_myprotocol.h, and the source file is called routing_myprotocol.cpp.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems, even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, routing_protocol.h, should contain the following:

- Prototypes for interface functions in source file, routing_myprotocol.cpp
- Constant definitions
- Data structure definitions and data types: `struct` and `enum` declarations

The source file, routing_myprotocol.cpp, should contain the following:

- Statement to include the protocol's header file:

```
#include "routing_myprotocol.h"
```

- Statements to include standard library functions and other header files needed by the protocol source file. A typical protocol source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include "api.h"          // EXATA_HOME/include/api.h
#include "network_ip.h"
                  // EXATA_HOME/libraries/developer/src/network_ip.h.
```

- Protocol initialization function, MyprotocolInit
- Protocol event dispatcher function, MyprotocolHandleProtocolEvent
- Protocol packet handler function, MyprotocolHandleProtocolPacket
- Protocol finalization function, MyprotocolFinalize
- Additional protocol implementation functions

The file network_ip.cpp contains the IP initialization, routing initialization, event dispatcher, and finalization functions. These IP functions in turn call the routing protocol functions MyprotocolInit, MyprotocolHandleProtocolEvent and MyprotocolFinalize. Therefore, to make these protocol functions available to the IP functions, insert the following include statement in the file network_ip.cpp:

```
#include "routing_myprotocol.h"
```

4.4.5.3 Including MYPROTOCOL in List of Routing Protocols

Each node in EXata maintains a list of routing protocols running at the node. When a new Network Layer unicast routing protocol is added to EXata, it needs to be included in the list of Network Layer protocols. To do this, add the protocol name to the enumeration `NetworkRoutingProtocolType` defined in EXATA_HOME/include/network.h (see Section 4.4.3).

For our example protocol, add the entry `ROUTING_PROTOCOL_MYPROTOCOL` to `NetworkRoutingProtocolType` as shown in Figure 4-67.

```
typedef enum
{
    NETWORK_PROTOCOL_IP = 0,
    NETWORK_PROTOCOL_IPV6,
    NETWORK_PROTOCOL_MOBILE_IP,
    ...
    ROUTING_PROTOCOL_AODV6,
    ROUTING_PROTOCOL_DYMO,
    ROUTING_PROTOCOL_DYMO6,
    ...
    ROUTING_PROTOCOL_MYPROTOCOL;
    ROUTING_PROTOCOL_NONE // this must be the last one
} NetworkRoutingProtocolType;
```

FIGURE 4-67. Adding MYPROTOCOL to List of Network Layer Protocols

Note **Always add to the end of lists in header files.**

EXata provides for detailed traces of packets as they traverse the protocol stack at nodes in the network. A packet trace lists, among other information, the protocol that is handling the packet at the time of the trace. To facilitate tracing, EXata lists all protocols in an enumeration, `TraceProtocolType`, in the file EXATA_HOME/include/trace.h. For our example protocol, add an entry `TRACE_MYPROTOCOL` in `TraceProtocolType`, as shown in Figure 4-68.

```
typedef enum
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,              // 1
    TRACE_UDP,                  // 2
    TRACE_IP,                   // 3
    ...
    TRACE_MYPROTCOL,
    // Must be last one!!!
    TRACE_ANY_PROTOCOL
}TraceProtocolType;
```

FIGURE 4-68. Adding MYPROTOCOL to List of Trace Protocols

Note **Always add to the end of lists in header files (just before the entry `TRACE_ANY_PROTOCOL`).**

A routing administrative distance is assigned to each routing protocol, which determines its priority relative to other routing protocols. A protocol with a lower administrative distance has a higher priority. The administrative distances of all routing protocols are defined in the enumeration `NetworkRoutingAdminDistanceType` defined in network.h (see Section 4.4.3).

For our example protocol, add the entry `ROUTING_ADMIN_DISTANCE_MYPROTOCOL` to `NetworkRoutingAdminDistanceType` as shown in Figure 4-69. Add this entry in the proper place in the list to reflect the desired priority of MYPROTOCOL relative to the other routing protocols.

```
typedef enum
{
    ROUTING_ADMIN_DISTANCE_STATIC = 1,
    ROUTING_ADMIN_DISTANCE_EBGPv4 = 20,
    ...
    ROUTING_ADMIN_DISTANCE_OLSR,
    ROUTING_ADMIN_DISTANCE_EIGRP,
    ROUTING_ADMIN_DISTANCE_MYPROTOCOL,
    //StartRIP
    ROUTING_ADMIN_DISTANCE_RIP,
    //EndRIP
    ...
    // Should always have the highest administrative distance
    // (i.e., least important).
    ROUTING_ADMIN_DISTANCE_DEFAULT = 255
} NetworkRoutingAdminDistanceType;
```

FIGURE 4-69. Declaring Administrative Distance for MYPROTOCOL

4.4.5.4 Defining Data Structures

Each routing protocol has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Protocol parameters (see Section 4.4.5.5.3.1)
2. Protocol state (see Section 4.4.5.5.3.2)
3. Statistics variables (see Section 4.4.5.10.1)
4. Routing table (see Section 4.4.5.5.3.2)

Define an appropriate data structure for MYPROTOCOL called `MyprotocolData` in the protocol header file, routing_protocol.h. As an example, the following data structure (defined in routing_aodv.h) is used by the AODV protocol:

```
typedef struct struct_network_aodv_str
{
    // set of user configurable parameters
    Int32        netDiameter;
    clocktype    nodeTraversalTime;
    clocktype    myRouteTimeout;
    ...
    // set of aodv protocol dependent parameters
    AodvRoutingTable routeTable;
    ...
    AodvStats stats;
    BOOL statsCollected;
    BOOL statsPrinted;
    BOOL processHello;
    BOOL processAck;
    ...
    BOOL isExpireTimerSet;
    BOOL isDeleteTimerSet;
    ...
} AodvData;
```

In the above declaration, `AodvRoutingTable` is the data structure for the AODV routing table and `AodvStats` is the data structure for AODV statistics (see Section 4.4.5.10.1).

4.4.5.5 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a Network Layer routing protocol.

4.4.5.5.1 Determining the Protocol Configuration Format

A routing protocol may use protocol-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a routing protocol's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

As an example, the following are some of the configuration parameters for the AODV protocol. Refer to file EXATA_HOME/scenarios/default/default.config for an explanation of these parameters.

```
AODV-ACTIVE-ROUTE-TIMEOUT    400MS
AODV-RREQ-RETRIES            3
AODV-LOCAL-REPAIR            YES
```

A configuration variable is not always mandatory. If an optional configuration variable is not assigned a value in the configuration file, the default value is used. For example, if a user does not specify a value for `AODV-ACTIVE-ROUTE-TIMEOUT`, the default value of 300 milliseconds is used by the protocol.

Decide on the format for specifying the new routing protocol's configuration parameters. Section 4.4.5.5.3.1 explains how to read user input specified in this format to initialize the routing protocol.

4.4.5.5.2 Calling the Protocol Initialization Function

The protocol stack of each node is initialized in a bottom up manner. The initialization of the Network Layer thus occurs after the layers below it have been initialized. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the Network Layer initialization function NETWORK_Initialize, which is implemented in the file EXATA_HOME/main/network.cpp. Function NETWORK_Initialize, in turn, calls the IP initialization function NetworkIpInit and the routing initialization function IpRoutingInit, which are implemented in the file network_ip.cpp. Function NetworkIpInit in turn calls function NetworkIpParseAndSetRoutingProtocolType, which reads the name of the routing protocol for each interface from the configuration file and updates the routing protocol type for that interface. Function IpRoutingInit calls the initialization function of the routing protocol configured on the interface. The code snippets from NetworkIpParseAndSetRoutingProtocolType and IpRoutingInit corresponding to AODV is shown are Figure 4-70 and Figure 4-71, respectively. The functions used in the example are explained below.

- Function IO_ReadString reads the name of the routing protocol from the configuration file. The prototype for IO_ReadString is defined in EXATA_HOME/include/fileio.h.
- Function NetworkIpGetInterfaceAddress, defined in network_ip.cpp, returns the IP address associated with an interface.
- Function NetworkIpAddUnicastRoutingProtocolType, defined in network_ip.cpp, initializes the routing protocol information for an interface. In the example of Figure 4-70, NetworkIpAddUnicastRoutingProtocolType updates the `IpInterfaceInfoType` structure associated with the interface (see Section 4.4.3) by setting the `routingProtocolType` field to `ROUTING_PROTOCOL_AODV` and the `routingProtocol` field to `NULL`.
- Function NetworkIpGetRoutingProtocol, defined in network_ip.cpp, returns a pointer to the data structure associated with the specified routing protocol. If the same routing protocol is running at multiple interfaces of a node, a single instance of the data structure for the routing protocol is shared by all interfaces. If the routing protocol is running at one of the interfaces and that interface has been assigned a routing protocol structure, NetworkIpGetRoutingProtocol returns a pointer to that structure; otherwise, it returns `NULL`. In the example of Figure 4-71, NetworkIpGetRoutingProtocol returns a pointer to the structure `AodvData` or `NULL`.
- Function AodvInit, defined in routing_aodv.cpp, is the initialization function for AODV. AodvInit is called if function NetworkIpGetRoutingProtocol returns `NULL`, i.e., an instance of `AodvData` is not associated with any interface. In addition to performing other initializing tasks (see Section 4.4.5.5.3), AodvInit creates an instance of the AODV data structure, `AodvData`, and associates it with the specified

interface by updating the routingProtocol field of the IpInterfaceInfoType structure associated with the interface to point to the newly created instance of AodvData.

- Function NetworkIpUpdateUnicastRoutingProtocolAndRouterFunction, defined in network_ip.cpp, is called if function NetworkIpGetRoutingProtocol returns a non-NULL pointer, i.e., if AODV is running at another interface and an instance of AodvData has been associated with that interface. NetworkIpUpdateUnicastRoutingProtocolAndRouterFunction associates the same instance of AodvData with the specified interface. This ensures that even if a routing protocol is running at multiple interfaces of a node, all interfaces running the same routing protocol share one instance of the protocol data structure.

```
void
NetworkIpParseAndSetRoutingProtocolType (Node *node,
                                         const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    BOOL retVal;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
         ...
         IO_ReadString(
                node->nodeId,
                NetworkIpGetInterfaceAddress(node, i),
                nodeInput,
                "ROUTING-PROTOCOL",
                &retVal,
                protocolString);

        if (retVal)
        {
            ...
            else if (strcmp(protocolString, "AODV") == 0)
            {
                routingProtocolType = ROUTING_PROTOCOL_AODV;
            }
            ...
        }

      NetworkIpAddUnicastRoutingProtocolType(
                        node,
                        routingProtocolType,
                        i,
                        NETWORK_IPV4);
     }
     ...
}
```

FIGURE 4-70. Initializing Routing Protocol Information for an Interface

```
void
IpRoutingInit(Node *node,
              const NodeInput *nodeInput)
{
     ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
        if (NetworkIpGetInterfaceType(node, i) == NETWORK_IPV4
            || NetworkIpGetInterfaceType(node, i) == NETWORK_DUAL)
        {
           switch (ip->interfaceInfo[i]->routingProtocolType)
           {
              ...
              case ROUTING_PROTOCOL_AODV:
              {
                if (!NetworkIpGetRoutingProtocol(node,
                                        ROUTING_PROTOCOL_AODV))
                {
                   AodvInit(
                      node,
                      (AodvData**)&ip->interfaceInfo[i]->routingProtocol,
                      nodeInput,
                      i,
                      ROUTING_PROTOCOL_AODV);
                }
                else
                {
                  NetworkIpUpdateUnicastRoutingProtocolAndRouterFunction(
                                          node,
                                          ROUTING_PROTOCOL_AODV,
                                          i);
                }
                break;
              }
              ...
           }
        }
     }
  ...
}
```

FIGURE 4-71. Calling AODV Initialization Function from Routing Initialization Function

Figure 4-72 shows the modifications to be made to IpRoutingInit to incorporate MYPROTOCOL in EXata. MyprotocolInit is the initialization function (see Section 4.4.5.5.3) and `MyprotocolData` is the protocol data structure (see Section 4.4.5.4) for MYPROTOCOL.

```
void
IpRoutingInit(Node *node, const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
        if (NetworkIpGetInterfaceType(node, i) == NETWORK_IPV4
            || NetworkIpGetInterfaceType(node, i) == NETWORK_DUAL)
        {
            switch (ip->interfaceInfo[i]->routingProtocolType)
            {
               ...
                case ROUTING_PROTOCOL_AODV:
                {
                    ...
                }
                ...
                case ROUTING_PROTOCOL_MYPROTOCOL:
                {
                    if (!NetworkIpGetRoutingProtocol(node,
                        ROUTING_PROTOCOL_MYPROTOCOL))
                    {
                        MyprotocolInit(
                                node,
                             (MyprotocolData **)
                                       &ip->interfaceInfo[i]->routingProtocol,
                          nodeInput,
                          i,
                          ROUTING_PROTOCOL_MYPROTOCOL);
                    }
                    else
                    {
                       NetworkIpUpdateUnicastRoutingProtocolAndRouterFunction(
                            node,
                            ROUTING_PROTOCOL_MYPROTOCOL,
                            i);
                    }
                    break;
                }
        ...
     }
     ...
}
```

FIGURE 4-72. Calling MYPROTOCOL Initialization Function from IP Initialization Function

4.4.5.5.3 Implementing the Protocol Initialization Function

The initialization of a Network Layer routing protocol takes place in the initialization function of the protocol that is called by the routing initialization function IpRoutingInit (see Figure 4-71). The initialization function of a routing protocol commonly performs the following tasks:

- Create an instance of the protocol data structure
- Read and store the user-specified configuration parameters
- Initialize the state variables and routing table
- Register the protocol's router function and other callback functions with IP
- Schedule a timer to itself for starting the protocol

Like all other functions belonging to the protocol, the prototype for the initialization function, MyprotocolInit, should be included in the protocol's header file, routing_protocol.h.

4.4.5.5.3.1 Creating an Instance and Reading Configuration Parameters

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as flags, connection information, routing table used by the protocol, etc.

To store the state, declare the structure to hold the protocol state in the header file, routing_protocol.h (see Section 4.4.5.4). As an example, see the declaration of the AODV data structure `AodvData` in routing_aodv.h.

Create an instance of the protocol state by allocating memory to the state structure. AODV performs this task in its initialization function AodvInit by calling the function MEM_malloc to allocate memory for the AODV data structure `AodvData`, as shown in Figure 4-73. AodvInit and the other AODV functions are implemented in routing_aodv.cpp. Data structure and constant definitions for AODV are contained in routing_aodv.h.

The next step is to read the user-defined configuration parameters from the input file and store them in the protocol data structure. AodvInit does this by calling function AodvInitializeConfigurableParameters. AodvInitializeConfigurableParameters, shown in Figure 4-74, uses IO functions such as IO_ReadTime and IO_ReadString to read parameter values from the input file and store them in the appropriate fields of the protocol data structure `AodvData`. If a value is not specified for a parameter in the input file, AodvInitializeConfigurableParameters stores the default value for that parameter. IO_ReadTime, IO_ReadString and other IO functions are defined in EXATA_HOME/include/fileio.h.

```
void AodvInit(Node* node,AodvData** aodvPtr,const NodeInput* nodeInput,
              int interfaceIndex, NetworkRoutingProtocolType aodvProtocolType)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar
    AodvData* aodv = (AodvData *) MEM_malloc(sizeof(AodvData));
    ...
    (*aodvPtr) = aodv;
    ...
    // Read whether statistics needs to be collected for the protocol
    ...
    // Initialize statistical variables
    ...
    // Read user configurable parameters from the configuration file or
    // initialize them with the default values.
    AodvInitializeConfigurableParameters(
        node,
        nodeInput,
        aodv,
        aodv->iface[interfaceIndex].address);
   // Initialize AODV routing table
   ...
   if(aodv->iface[interfaceIndex].ip_version == NETWORK_IPV4)
    {
        // Set the mac status handler function
        NetworkIpSetMacLayerStatusEventHandlerFunction(
            node, &Aodv4MacLayerStatusHandler, interfaceIndex);
        // Set the router function
        NetworkIpSetRouterFunction(
            node,
            &Aodv4RouterFunction,
            interfaceIndex);
        destAddr.networkType = NETWORK_IPV4;
        destAddr.interfaceAddr.ipv4 = ANY_DEST;
        protocolType = ROUTING_PROTOCOL_AODV;
        // Set default Interface Info
        aodv->defaultInterface = interfaceIndex;
        SetIPv4AddressInfo(
            &aodv->defaultInterfaceAddr,
            NetworkIpGetInterfaceAddress(node, interfaceIndex));
    }
    else if(aodv->iface[interfaceIndex].ip_version == NETWORK_IPV6)
    {
       ...
     }

    if (aodv->processHello)
    {
       ...
        AodvSetTimer(node, MSG_NETWORK_SendHello, destAddr,
                     AODV_HELLO_INTERVAL);
    }
}
```

FIGURE 4-73. AODV Initialization Function

```
static
void AodvInitializeConfigurableParameters(
    Node* node,
    const NodeInput* nodeInput,
    AodvData* aodv,
    Address interfaceAddress)
{
    BOOL wasFound;
    char buf[MAX_STRING_LENGTH];
    UInt32 nodeId = node->nodeId;
    ...
    IO_ReadTime(
        nodeId,
        &interfaceAddress,
        nodeInput,
        "AODV-ACTIVE-ROUTE-TIMEOUT",
        &wasFound,
        &aodv->activeRouteTimeout);

    if (!wasFound)
    {
        aodv->activeRouteTimeout = AODV_DEFAULT_ACTIVE_ROUTE_TIMEOUT;
    }
    else
    {
        ERROR_Assert(
                aodv->activeRouteTimeout > 0,
        "Invalid AODV_DEFAULT_ACTIVE_ROUTE_TIMEOUT configuration");
    }
    ...
    IO_ReadInt(
        nodeId,
        &interfaceAddress,
        nodeInput,
        "AODV-RREQ-RETRIES",
        &wasFound,
        &aodv->rreqRetries);

    if (!wasFound)
    {
        aodv->rreqRetries = AODV_DEFAULT_RREQ_RETRIES;
    }
    else
    {
        ERROR_Assert(
                aodv->helloInterval > 0,
        "Invalid AODV_DEFAULT_HELLO_INTERVAL configuration");
    }
   ...
}
```

FIGURE 4-74. Reading AODV Configuration Parameters

4.4.5.5.3.2 Initializing State Variables and Routing Table

The initialization function of a routing protocol also initializes the state variables and routing table used by the protocol. Figure 4-75 shows code segments to initialize the state variables and routing table for AODV.

A routing protocol may need to maintain a list of neighbors. In this case, the list of neighbors should be initialized in the initialization function of the protocol.

```
void
AodvInit(
    Node* node,
    AodvData** aodvPtr,
    const NodeInput* nodeInput,
    int interfaceIndex,
    NetworkRoutingProtocolType aodvProtocolType)
{
    ...

   // Initialize AODV routing table
    for (i = 0; i < AODV_ROUTE_HASH_TABLE_SIZE; i++)
    {
        (&aodv->routeTable)->routeHashTable[i] = NULL;
    }

    (&aodv->routeTable)->routeExpireHead = NULL;
        ...
    // Initialize aodv structure to store RREQ information
    (&aodv->seenTable)->front = NULL;
    ...
    // Initialize Aodv sequence number
    aodv->seqNumber = 0;

    // Initialize Aodv Broadcast id
    aodv->floodingId = 0;

    // Initialize Last Broadcast sent
    aodv->lastBroadcastSent = (clocktype) 0;
    ...
}
```

FIGURE 4-75. Initializing AODV State Variables and Routing Table

4.4.5.5.3.3 Registering Callback Functions with IP

A Network Layer routing protocol interacts with IP to route packets and to handle protocol events. To do this efficiently, the routing protocol registers the functions that perform these tasks with IP as part of initialization, by passing pointers to these functions to IP. These functions are called *callback functions*. When IP encounters an event that needs to be handled by the routing protocol, IP can directly call the appropriate callback function that processes that event. A routing protocol should register with IP its router function and other callback functions that handle events that are of interest to the protocol.

IP callback functions implemented in EXata and the API functions used to register them are listed below. See network_ip.cpp for a description of parameters of these functions.

1. **Callback Function**: Router function used by the protocol

   API to Register Function: NetworkIpSetRouterFunction

   Function Type: RouterFunctionType
2. **Callback Function**: Function to handle MAC Layer status changes

   API to Register Function: NetworkIpSetMacLayerStatusEventHandlerFunction

   Function Type: MACLayerStatusEventHandlerFunctionType
3. **Callback Function**: Function to promiscuously peek at packets not addressed to the node

   API to Register Function: NetworkIpSetPromiscuousMessagePeekFunction

   Function Type: PromiscuousMessagePeekFunctionType
4. **Callback Function**: Function to handle MAC Layer acknowledgements

   API to Register Function: NetworkIpSetMacLayerAckHandler

   Function Type: MacLayerAckHandlerType
5. **Callback Function**: Function to handle route update events

   API to Register Function: NetworkIpSetRouteUpdateEventFunction

   Function Type: NetworkRouteUpdateEventType

As an example, the AODV initialization function AodvInit (see Figure 4-73) calls function NetworkIpSetRouterFunction to register with IP the router function used by AODV when operating with Ipv4, Aodv4RouterFunction. This enables IP to directly call Aodv4RouterFunction to determine the route for a packet if AODV is running at that interface. Similarly, AodvInit calls function NetworkIpSetMacLayerStatusEventhandlerFunction to register with IP the MAC Layer status handler function used by AODV when operating with IPv4, Aodv4MacLayerStatusHandler. This enables IP to directly call Aodv4MacLayerStatusHandler to handle a MAC Layer status change, if AODV is running at that interface.

4.4.5.5.3.4 Initializing Timers

A routing protocol may need to set a timer at initialization. For example, if *Hello* messages are enabled in AODV, function AodvInit calls function AodvSetTimer to set the initial timer (see Figure 4-73). See Section 3.3.2.2 for details on setting timers.

4.4.5.6 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a Network Layer routing protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the Network Layer, NODE_ProcessEvent calls the Network Layer event dispatcher NETWORK_ProcessEvent, defined in EXATA_HOME/main/network.cpp. If IP is running at the Network Layer, NETWORK_ProcessEvent calls the IP event dispatcher NetworkIpLayer, defined in network_ip.cpp.

Section 4.4.5.6.1 describes how to modify the IP event dispatcher function to call the routing protocol's event dispatcher. Section 4.4.5.6.2 describes how to implement the routing protocol's event dispatcher.

4.4.5.6.1 Modifying the IP Event Dispatcher

Function NetworkIpLayer implements the IP event dispatcher that informs the appropriate Network Layer protocol of received events. Messages contain the name of the protocol they are destined for. (This is the routing protocol name specified in the enumerated data type `NetworkRoutingProtocolType`, described in Section 4.4.5.3.) NetworkIpLayer implements a switch statement on the protocol name read from the message and calls the appropriate protocol-specific event dispatcher.

To enable the protocol MYPROTOCOL to receive events, add code to NetworkIpLayer to call the protocol's event dispatcher function when messages for the protocol are received. Figure 4-76 shows a code fragment from NetworkIpLayer with sample code for calling MYOPROTOCOL's event dispatcher function MyprotocolHandleProtocolEvent.

```
void
NetworkIpLayer(Node *node, Message *msg)
{
    switch (msg->protocolType)
    {
        case GROUP_MANAGEMENT_PROTOCOL_IGMP:
        {
            IgmpLayer(node, msg);
            break;
        }
        ...
        case ROUTING_PROTOCOL_AODV:
        {
            AodvHandleProtocolEvent(node, msg);
            break;
        }
        ...
        case ROUTING_PROTOCOL_MYPROTOCOL:
        {
            MyprotocolHandleProtocolEvent(node, msg);
            break;
        }
        ...
 }//switch//
}
```

FIGURE 4-76. IP Event Dispatcher

4.4.5.6.2 Implementing the Protocol Event Dispatcher

A routing protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received.

All event types used by EXata protocols are enumerated in the file EXATA_HOME/include/api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as shown in Figure 4-77.

```
// /**
// ENUM        :: MESSAGE/EVENT
// DESCRIPTION :: Event/message types exchanged in the simulation
// **/
enum
{
    /* Special message types used for internal design. */
    MSG_SPECIAL_Timer                          = 0,
    ...
    /* Message Types for Channel layer */
    MSG_PROP_SignalArrival                     = 100,
    MSG_PROP_SignalEnd                         = 101,
    ...
    /*
     * Any other message types which have to be added should be added before
     * MSG_DEFAULT. Otherwise the program will not work correctly.
     */
    MSG_NETWORK_MYPROTOCOL_NewEvent1,
    MSG_NETWORK_MYPROTOCOL_NewEvent2,
    MSG_DEFAULT                                = 10000
};
```

FIGURE 4-77. Declaring New Event Types

Note **Always add to the end of lists in header files (just before the entry `MSG_DEFAULT`).**

The event dispatcher function for a routing protocol generally handles timer events. (Packet events are handled by a separate packet handler, as discussed in Section 4.4.5.8.2.) As an example, Figure 4-78 shows the AODV event dispatcher function AodvHandleProtocolEvent. See files routing_aodv.h and routing_aodv.cpp for definitions of data structures and functions used for implementing AODV.

Note that once a message has been processed, it is freed by calling the API MESSAGE_Free, unless it is used to reset the timer for a future time. The event dispatcher also includes a default case in the switch statement to handle events of an unknown type.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

As part of handling an event, a routing protocol at a node may transmit packets to its peers using MAC Layer services. EXata provides the API functions listed below that enable a Network Layer protocol to send packets to the MAC Layer. These functions are implemented in the file network_ip.h. See *API Reference Guide* or the file network_ip.h for an explanation of these functions and their parameters.

1. NetworkIpSendPacketOnInterfaceWithDelay
2. NetworkIpSendRawMessage
3. NetworkIpSendRawMessageWithDelay
4. NetworkIpSendRawMessageToMacLayer
5. NetworkIpSendRawMessageToMacLayerWithDelay
6. NetworkIpSendPacketToMacLayer
7. NetworkIpSendPacketToMacLayerWithDelay
8. NetworkIpSendPacketOnInterface
9. NetworkIpSendPacketOnInterfaceWithDelay
10. NetworkIpSendPacketToMacLayerWithNewStrictSourceRoute

```
void
AodvHandleProtocolEvent(
    Node* node,
    Message* msg)
{
    AodvData* aodv = NULL;

    if (MESSAGE_GetProtocol(msg) == ROUTING_PROTOCOL_AODV6)
    {
        aodv = (AodvData *) NetworkIpGetRoutingProtocol(
                                node,
                                ROUTING_PROTOCOL_AODV6,
                                NETWORK_IPV6);
    }
    else
    {
        aodv = (AodvData *) NetworkIpGetRoutingProtocol(
                                node,
                                ROUTING_PROTOCOL_AODV,
                                NETWORK_IPV4);
    }
    switch (MESSAGE_GetEvent(msg))
    {
        // Remove an entry from the RREQ Seen Table
        case MSG_NETWORK_FlushTables:
        {
            ...
            AodvDeleteSeenTable(&aodv->seenTable);
            MESSAGE_Free(node, msg);
            break;
        }
        // Check connectivity based on hello msg
        case MSG_NETWORK_CheckNeighborTimeout:
        {
            ...
        }
        // Remove the route that has not been used for awhile
        case MSG_NETWORK_CheckRouteTimeout:
        {
            ...
        }
        case MSG_NETWORK_DeleteRoute:
        {
            ...
        }
        ...
        default:
        {
            ...
        }
    }
}
```

FIGURE 4-78. AODV Event Dispatcher

4.4.5.7 Modifying IP Functions

The IP function NetworkRoutingGetAdminDistance, implemented in network_ip.cpp, returns the administrative distance of a routing protocol (see Section 4.4.3). This function should be modified to enable IP to determine the routing distance (i.e., the relative priority) of MYPROTOCOL, if MYPROTOCOL uses the IP forwarding table. Figure 4-79 shows the modifications that need to be made to NetworkRoutingGetAdminDistance to add MYPROTOCOL.

```
NetworkRoutingAdminDistanceType
NetworkRoutingGetAdminDistance(
    Node *node,
    NetworkRoutingProtocolType type)
{
    switch (type)
    {
        case ROUTING_PROTOCOL_STATIC:
        {
            return ROUTING_ADMIN_DISTANCE_STATIC;
        }
        ...
        case ROUTING_PROTOCOL_BELLMANFORD:
        {
            return ROUTING_ADMIN_DISTANCE_BELLMANFORD;
        }
        case ROUTING_PROTOCOL_MYPROTOCOL:
        {
            return ROUTING_ADMIN_DISTANCE_MYPROTOCOL;
        }
        ...
    }
}
```

FIGURE 4-79. Modifications to Function NetworkRoutingGetAdminDistance

4.4.5.8 Processing Routing Packets

A routing protocol running at the Network Layer of a node exchanges *routing packets* with other nodes to maintain routing information. Routing packets are control packets that carry information for the routing protocol. These routing packets are different from *data packets*, which carry user data. Data packets are received at the Network Layer from the Transport Layer or from other nodes. Data packets received from other nodes that are addressed to the node are delivered to the Transport Layer. Data packets received from the Transport Layer and data packets received from other nodes that are not addressed to the node are sent out on the interface determined by the packets' destination address and the routing information maintained by the routing protocol.

In this section, we describe the steps for implementing the function that processes routing packets of a Network Layer routing protocol. Section 4.4.5.8.1 describes how to modify the IP packet handling function to call the protocol's routing packet handler. Section 4.4.5.8.2 describes how to implement the protocol's routing packet handler.

4.4.5.8.1 Modifying IP Packet Handler

The IP function DeliverPacket, defined in network_ip.cpp, implements the packet handler for IP. DeliverPacket sends packets received from the MAC layer to the appropriate Network Layer or Transport Layer protocol based upon the IP protocol number contained in the IP header of the received packet. To add a new routing protocol, MYPROTOCOL, at the Network Layer, assign an IP protocol number to MYPROTOCOL and modify function DeliverPacket to deliver packets to MYPROTOCOL.

1. Define an IP Protocol Number for MYPROTOCOL. File network_ip.h contains constant definitions for all Transport Layer and Network Layer protocols (see Figure 4-80). For example, the IP Protocol Number for AODV is 123. Add a constant definition to associate an IP Protocol Number with MYPROTOCOL.

   Note **Be sure to use an IP Protocol Number that is not already used for some other protocol.**

```
//--------------------------------------------------------
// IP protocol numbers
//--------------------------------------------------------

// IP protocol numbers for network-layer and transport-layer protocols

...
// /**
// CONSTANT    :: IPPROTO_AODV : 123
// DESCRIPTION :: IP protocol numbers for AODV.
// **/
#define IPPROTO_AODV            123

// /**
// CONSTANT    :: IPPROTO_DSR : 135
// DESCRIPTION :: IP protocol numbers for DSR.
// **/
...
// /**
// CONSTANT    :: IPPROTO_MYPROTOCOL : 255
// DESCRIPTION :: IP protocol number for MYPROTOCOL.
// **/
#define IPPROTO_MYPROTOCOL      255
...
```

FIGURE 4-80. Declaring IP Protocol Number for MYPROTOCOL

2. Function DeliverPacket, shown in Figure 4-81, performs a switch on the IP Protocol Number, `ipProtocolNumber`, contained in the IP header of the received packet, and calls the appropriate routine to deliver the packet to the protocol identified by `ipProtocolNumber`. If `ipProtocolNumber` corresponds to a unicast routing protocol running at the Network Layer, DeliverPacket checks if that routing protocol is running at the specified interface by calling function NetworkIpGetUnicastRoutingProtocolType. If `ipProtocolNumber` corresponds to the routing protocol running at that interface, DeliverPacket calls the packet handler function for that protocol. For example, DeliverPacket calls function AodvHandleProtocolPacket to deliver a packet to AODV if the IP Protocol Number read from the packet's header is 123 and AODV is running at that interface.

   Add code to DeliverPacket to call MYPROTOCOL's packet handler function, MyprotocolHandleProtocolPacket, with appropriate parameters, if the IP Protocol Number read from the packet's header is `IPPROTO_MYPROTOCOL` (defined in step 1) as shown in Figure 4-81.

```
static void //inline//
DeliverPacket(Node *node, Message *msg,
              int interfaceIndex, NodeAddress previousHopAddress)
{
    ...
    switch (ipProtocolNumber)
    {
        ...
        // Delivery to network-layer routing protocols.
        ...
        case IPPROTO_AODV:
        {
            if (NetworkIpGetUnicastRoutingProtocolType(node, interfaceIndex) ==
                ROUTING_PROTOCOL_AODV)
            {
                Address srcAddress;
                Address destAddress;
                SetIPv4AddressInfo(&srcAddress, sourceAddress);
                SetIPv4AddressInfo(&destAddress, destinationAddress);
                AodvHandleProtocolPacket(node, msg, srcAddress, destAddress,
                                         ttl, interfaceIndex);
            }
            else
            {
                //Trace drop
                ...
                MESSAGE_Free(node, msg);
            }
            break;
        }
        case IPPROTO_MYPROTOCOL:
        {
            if (NetworkIpGetUnicastRoutingProtocolType(node, interfaceIndex) ==
                ROUTING_PROTOCOL_MYPROTOCOL)
            {
                // Write Code similar to AODV.
                // Call MyprotocolHandleProtocolPacket here
            }
            else
            {
                MESSAGE_Free(node, msg);
            }
            break;
        }
        ...
    }//switch//
}//DeliverPacket//
```

FIGURE 4-81. Delivering Packets from IP to Network Layer Routing Protocols

4.4.5.8.2 Implementing the Protocol Packet Handler

A routing protocol's packet handler should include a switch on all types of packets that the protocol may receive. It can then process each packet type either inside the switch or by calling a function to handle the packet type received. For example, function AodvHandleProtocolPacket, shown in Figure 4-82 and implemented in routing_aodv.cpp, processes AODV routing packets.

As part of processing a received packet, a routing protocol at a node may transmit packets to its peers. See Section 4.4.5.6.2 for details of sending packets.

Write function MyprotocolHandleProtocolPacket to handle routing packets for MYPROTOCOL. Follow the example of AodvHandleProtocolPacket or the packet handler for some other Network Layer unicast routing protocol in EXata.

```
void
AodvHandleProtocolPacket(
    Node* node,
    Message* msg,
    NodeAddress srcAddr,
    NodeAddress destAddr,
    int ttl,
    int interfaceIndex)
{
    UInt32* packetType = (UInt32* )MESSAGE_ReturnPacket(msg);
    BOOL IPV6 = FALSE;

    if(srcAddr.networkType == NETWORK_IPV6)
    {
        IPV6 = TRUE;
    }

    if (AODV_DEBUG_AODV_TRACE)
    {
        AodvPrintTrace(node, msg, 'R', IPv6);
    }

    switch (*packetType >> 24)
    {
        case AODV_RREQ:
        {
            ...
            AodvHandleRequest(
                node,
                msg,
                srcAddr,
                ttl,
                interfaceIndex);

            MESSAGE_Free(node, msg);
            break;
        }
        ...
        default:
        {
           ERROR_Assert(FALSE, "Unknown packet type for Aodv");
           break;
        }
    }
}
```

FIGURE 4-82. AODV Routing Packet Handling Function

4.4.5.9 Implementing Callback Functions

As explained in Section 4.4.5.5.3.3, a Network Layer routing protocol in EXata implements certain callback functions, including a router function to determine routes. When IP receives a data packet from the Transport Layer, it calls function RoutePacketAndSendToMac to determine the next hop and outgoing interface for the packet. Similarly, when IP receives a data packet from the MAC Layer that needs to be forwarded to another node, IP calls RoutePacketAndSendToMac to determine the next hop and outgoing interface for the packet. Function RoutePacketAndSendToMac calls the router function of the routing protocol running at the interface on which the packet arrives (see Figure 4-83).

```
void
RoutePacketAndSendToMac(Node *node,
                        Message *msg,
                        int incomingInterface,
                        int outgoingInterface,
                        NodeAddress previousHopAddress)
{
 ...
 if (!packetWasRouted)
        {
            RouterFunctionType routerFunction = NULL;
            routerFunction = NetworkIpGetRouterFunction(node,
                                                        interfaceIndex);
            if (routerFunction)
            {
                (routerFunction)(node,
                                 msg,
                                 ipHeader->ip_dst,
                                 previousHopAddress,
                                 &PacketWasRouted);
            }
            if (!packetWasRouted)
            {
                if (IpHeaderHasSourceRoute(ipHeader))
                {
                    SourceRouteThePacket(node, msg);
                }
                else
                {
                    RouteThePacketUsingLookupTable(node,
                                                   msg,
                                                   incomingInterface);
                }//if//
            }//if//
        }//if//
    }//if//
}//RoutePacketAndSendToMac//
```

FIGURE 4-83. Calling the Router Function

The details of the router function depend upon the routing algorithm used by the protocol. As an example, function AodvRouterFunction, defined in routing_aodv.cpp, is the router function for AODV. The prototype for the function AodvRouterFunction is shown below:

```
void AodvRouterFunction(
    Node *node,
    Message *msg,
    Address destAddr,
    Address previousHopAddress,
    BOOL *packetWasRouted) {}
```

Another example of a callback function is the MAC Layer status handler function. Some Network Layer routing protocols interact with the MAC Layer and update their routing information when the status of the MAC Layer changes. AODV is an example of such a protocol. AODV implements a MAC Layer status handler function, AodvMacLayerStatusHandler, to handle MAC Layer status changes. AodvMacLayerStatusHandler is defined in routing_aodv.cpp.

Write the router function for MYPROTOCOL. Write any other callback functions that MYPROTOCOL may need. Use AODV or some other appropriate routing protocol as an example. Register all callback functions with IP in the protocol's initialization function (see Section 4.4.5.5.3.3).

4.4.5.10 Collecting and Reporting Statistics

In this section, we describe how to collect and report statistics for a Network Layer routing protocol.

4.4.5.10.1 Declaring Statistics Variables

A Network Layer routing protocol can be configured to record statistics specified by the programmer, such as:

- Number of request packets sent
- Number of reply packets received
- Number of data packets forwarded

To enable statistics collection for the protocol, include the statistic collection variables in the structure used to hold the protocol state (see Section 4.4.5.4). The statistics related variables can also be defined in a structure and then that structure is included in the state variable. For example, the data structure for AODV, `AodvData`, contains the AODV statistics variable, `AodvStats`, shown below:

```
typedef struct {
    D_UInt32 numRequestInitiated;
    UInt32 numRequestResent;
    UInt32 numRequestRelayed;
    ...
    UInt32 numReplyRecved;
    ...
    UInt32 numDataInitiated;
    UInt32 numDataForwarded;
    ...
} AodvStats;
```

`AodvData` and `AodvStats` are defined in routing_aodv.h.

4.4.5.10.2 Initializing Statistics

Initialize statistics variables in the protocol's initialization function. Determine whether statistics collection is enabled for the protocol and set the statistics collection flag accordingly. For example, field `statsCollected` of `AodvData` is a Boolean flag that indicates whether statistics collection is enabled for AODV. This flag is set to TRUE or FALSE by the initialization function AodvInit depending upon the input configuration, as shown in Figure 4-84. Function AodvInit allocates memory for the AODV data structure `AodvData`, which contains the AODV statistics structure `AodvStats`, and initializes all fields of `AodvStats` to 0.

```
void
AodvInit(
    Node* node,
    AodvData** aodvPtr,
    const NodeInput* nodeInput,
    int interfaceIndex,
    NetworkRoutingProtocolType aodvProtocolType)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    AodvData* aodv = (AodvData *) MEM_malloc(sizeof(AodvData));
    BOOL retVal;
    char buf[MAX_STRING_LENGTH];
    int i = 0;
    ...
    (*aodvPtr) = aodv;

    memset(aodv, 0, sizeof(AodvData));
    ...

    // Read whether statistics needs to be collected for the protocol
      IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "ROUTING-STATISTICS",
        &retVal,
        buf);

    if ((retVal == FALSE) || (strcmp(buf, "NO") == 0))
    {
        aodv->statsCollected = FALSE;
    }
    else if (strcmp(buf, "YES") == 0)
    {
        aodv->statsCollected = TRUE;
    }
    else
    {
        ERROR_ReportError("Needs YES/NO against STATISTICS");
    }
    ...
}
```

FIGURE 4-84. Initializing Statistics Variables for a Routing Protocol

4.4.5.10.3 Updating Statistics

After declaring and initializing the statistics variables, update their values during the protocol life cycle, as required. For example, AODV increments the value of `numReplyRecved` in function AodvHandleReply (see routing_aodv.cpp) every time AODV receives an AODV *reply* packet, as shown in Figure 4-85.

```
static
void AodvHandleReply(
         Node* node,
         Message* msg,
         Address srcAddr,
         int interfaceIndex,
         Address destAddr)
{
    AodvData* aodv = NULL;
    AodvRrepPacket* rrepPkt = NULL;
    ...
    Aodv6RrepPacket* rrep6Pkt = NULL;
    if(srcAddr.networkType == NETWORK_IPV6)
    {
        aodv = (AodvData *) NetworkIpGetRoutingProtocol(
                                    node,
                                    ROUTING_PROTOCOL_AODV6,
                                    NETWORK_IPV6);
        ...
    }
    else
    {
        aodv = (AodvData *) NetworkIpGetRoutingProtocol(
                                    node,
                                    ROUTING_PROTOCOL_AODV,
                                    NETWORK_IPV4);
        ...
    }
    ...
    aodv->stats.numReplyRecved++;
    ...
}
```

FIGURE 4-85. Updating AODV Statistics

4.4.5.10.4 Printing Statistics

As a final step towards statistics collection, create a function to print statistics. Call this function from the finalization function of the protocol, which is discussed in Section 4.4.5.11. Alternatively, statistics can be printed directly in the finalization function, as shown in Figure 4-87.

4.4.5.10.5 Adding Dynamic Statistics

Dynamic statistics are statistic variables whose values can be observed in the EXata GUI during the simulation. See Section 5.2.3 for adding dynamic statistics to a protocol. Refer to *EXata User's Guide* for details of viewing dynamic statistics during the simulation.

4.4.5.11 Finalization

The finalization function of the protocol is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

At the end of simulation, the finalization function for each protocol is called to print the protocol statistics. As discussed in Section 3.4.3, the finalization function is called hierarchically. The node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp, calls the finalization function for Network Layer, NETWORK_Finalize, defined in EXATA_HOME/main/network.cpp. If IP is running at the Network Layer, NETWORK_Finalize calls the IP finalization function, NetworkIpFinalize, defined in network_ip.cpp. NetworkIpFinalize calls the finalization function of the routing protocol running at each interface.

4.4.5.11.1 Modifying the IP Finalization Function

Call the finalization function of a Network Layer routing protocol from the IP finalization function, NetworkIpFinalize, defined in network_ip.cpp. Figure 4-86 shows the outline of code that needs to be added to NetworkIpFinalize. Function MyprotocolFinalize is the finalization function of the protocol MYPROTOCOL.

```
void
NetworkIpFinalize(Node *node)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    Scheduler *schedulerPtr = NULL;
    int i = 0;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
        ...
        switch (NetworkIpGetUnicastRoutingProtocolType(node, i))
        {
            case MULTICAST_PROTOCOL_STATIC:
            {
                RoutingMulticastStaticFinalize(node);
                break;
            }

            ...
            case ROUTING_PROTOCOL_AODV:
            {
               AodvFinalize(node, i, NETWORK_IPV4);
               break;
            }
            ...
            case ROUTING_PROTOCOL_MYPROTOCOL:
            {
                MyprotocolFinalize(node, ...);
                break;
            }
            ...
    }
    ...
}
```

FIGURE 4-86. Finalization Function for IP

4.4.5.11.2 Implementing the Protocol Finalization Function

Write the finalization function for protocol, MyprotocolFinalize. If statistics collection is enabled for MYPROTOCOL, call a function to print the protocol statistics (see Section 4.4.5.10.4), or add code directly to MyprotocolFinalize to print statistics. AODV follows the latter approach. Function AodvFinalize, shown in Figure 4-87 and implemented in routing_aodv.cpp, is the finalization function for AODV. Use AodvFinalize as a template to write MyprotocolFinalize.

Function AodvFinalize calls the C function sprintf to create a single string containing the statistic name and statistic value, and then calls function IO_PrintStat to print that string to a file. Function IO_PrintStat function, defined in EXATA_HOME/include/fileio.h, requires the following parameters:

- Node pointer: Pointer to the node reporting the statistics.
- Layer: String indicating the layer. Set this to `"Network"` for the Network Layer.
- Protocol: String indicating the protocol name.
- Interface address: Interface address. Set this to `ANY_DEST` for Network Layer routing protocols.
- Instance identifier: Instance identifier or port number. Set this to -1 if there is no instance identifier.
- Buffer: String containing the statistics.

```
void
AodvFinalize(Node* node, int i, NetworkType networkType)
{
    ...
    if (aodv->statsCollected && !aodv->statsPrinted)
    {
        aodv->statsPrinted = TRUE;

        sprintf(buf, "Number of RREQ Packets Initiated = %u",
                (unsigned short) aodv->stats.numRequestInitiated);
        IO_PrintStat(
            node,
            "Network",
            aodvVerBuf,
            ANY_DEST,
            -1,
            buf);
        ...
        sprintf(buf, "Number of RREP Packets Received = %u",
            aodv->stats.numReplyRecved);

        IO_PrintStat(
            node,
            "Network",
            aodvVerBuf,
            ANY_DEST,
            -1,
            buf);
        ...
   }
}
```

FIGURE 4-87. Finalization Function for AODV

4.4.5.12 Including and Compiling Files

The final step in integrating your routing protocol into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the routing protocol in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Developer library, then modify EXATA_HOME/libraries/developer/Makefile-common as shown in Figure 4-88. Recompile EXata after making the changes.

```
...
# common sources
#
DEVELOPER_SRCS = \
$(DEVELOPER_SRCDIR)/adaptation_aal5.cpp \
$(DEVELOPER_SRCDIR)/adaptation.cpp \
...
$(DEVELOPER_SRCDIR)/resource_manager_cbq.cpp \
$(DEVELOPER_SRCDIR)/route_atm.cpp \
$(DEVELOPER_SRCDIR)/routing_bellmanford.cpp \
$(DEVELOPER_SRCDIR)/routing_myprotocol.cpp \
$(DEVELOPER_SRCDIR)/routing_rip.cpp \
$(DEVELOPER_SRCDIR)/routing_ripng.cpp \
$(DEVELOPER_SRCDIR)/routing_static.cpp \
...
```

FIGURE 4-88. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.4.5.13 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.4.6 Adding a Network Layer Multicast Routing Protocol

This section provides an overview of the flow of a Network Layer multicast routing protocol and provides an outline for developing and adding a new Network Layer multicast routing protocol to EXata. It describes how to develop code components common to most routing protocols such as initializing, sending and receiving packets, determining routes, and collecting statistics.

A multicast routing protocol performs many of the tasks performed by a unicast routing protocol. In addition, a multicast routing protocol also has to perform group management functions. In general, a multicast routing protocol employs Internet Group Management Protocol (IGMP) utilities for group management. When developing a new multicast routing protocol, the user should also refer to Section 4.4.5, since many of the tasks are the same as for developing a unicast routing protocol.

We illustrate the process of adding a Network Layer multicast routing protocol by using as an example the implementation code for the PIM (Protocol Independent Routing) routing protocol. PIM operates in two modes, Dense Mode (PIM-DM) and Sparse Mode (PIM-SM). The header file for the PIM implementation (for both PIM-DM and PIM-SM) is EXATA_HOME/libraries/multimedia_enterprise/src/multicast_pim.h. The PIM implementation uses three source files, multicast_pim.cpp (for both PIM-DM and PIM-SM), multicast_pim_dm.cpp (for PIM-DM only), and multicast_pim_sm.cpp (for PIM-SM only) in EXATA_HOME/libraries/multimedia_enterprise/src. In this section, we use the PIM-DM implementation as an example and

use code segments from the PIM-DM implementation files throughout this section to illustrate different steps in writing a Network Layer multicast routing protocol. After understanding the discussed code segments, look at the complete code for PIM-DM to understand how a Network Layer multicast routing protocol is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a Network Layer multicast routing protocol, MYPROTOCOL, to EXata. For those steps that are similar to the steps for writing a Network Layer unicast routing protocol, we refer the reader to the appropriate subsection of Section 4.4.5. The steps that are different for multicast routing protocols are described in detail in subsequent sections.

1. Create header and source files. Modify the file network_ip.cpp to include the protocol's header file (see Section 4.4.6.1).
2. Include the protocol in the list of Network Layer protocols and trace protocols (see Section 4.4.6.2).
3. Define data structures for the protocol (see Section 4.4.6.3).
4. Decide on the format for the protocol-specific configuration parameters (see Section 4.4.5.5.1).
5. Call the protocol's initialization function from the routing initialization function, IpRoutingInit (see Section 4.4.6.4.2).
6. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Read and store the configuration parameters (see Section 4.4.6.4.3.1).
   b. Initialize the state variables and data structures (see Section 4.4.6.4.3.2).
   c. Register the protocol's callback functions with IP and IGMP (see Section 4.4.6.4.3.3).
   d. Initialize timers (see Section 4.4.6.4.3.4).
7. Call the protocol event dispatcher from the IP event dispatcher, NetworkIpLayer (see Section 4.4.6.5.1).
8. Declare any new event types used by the protocol in the header file EXATA_HOME/include/api.h (see Section 4.4.6.5.2).
9. Write the protocol event dispatcher (see Section 4.4.6.5.2).
10. Implement the protocol's routing packet handler.
    a. Define an IP Protocol Number for the protocol (see Section 4.4.6.6.1).
    b. Write a function to handle routing packets (see Section 4.4.6.6.2).
    c. Call the routing packet handler function from the IP function DeliverPacket (see Section 4.4.6.6.1).
11. Write the call back functions used by the protocol (see Section 4.4.6.7).
12. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.4.5.10.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.4.5.10.2).
    c. Update the statistics as appropriate (see Section 4.4.5.10.3).
    d. Write a function to print the statistics (see Section 4.4.5.10.4).
    e. Add dynamic statistics to the protocol, if desired (see Section 4.4.5.10.5) .
13. Call the protocol finalization function from the IP finalization function, NetworkIpFinalize (see Section 4.4.5.11.1).
14. Write the protocol finalization function (see Section 4.4.5.11.2). Call the function to print statistics from the protocol finalization function.
15. Include the protocol header and source files in the EXata tree and compile (see Section 4.4.5.12).
16. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.4.6.10).

4.4.6.1 Creating Files

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.2), except that in keeping with the naming guidelines of Section 4.4.5.1, the files for the example multicast routing protocol are called multicast_myprotocol.h and multicast_myprotocol.cpp.

4.4.6.2 Including MYPROTOCOL in List of Routing Protocols

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.3), except that there is no routing administrative distance associated with a multicast protocol.

For our example protocol, add the entry `MULTICAST_PROTOCOL_MYPROTOCOL` to `NetworkRoutingProtocolType`, defined in EXATA_HOME/include/network.h, as shown in Figure 4-89.

```
typedef enum
{
    NETWORK_PROTOCOL_IP = 0,
    NETWORK_PROTOCOL_IPV6,
    NETWORK_PROTOCOL_MOBILE_IP,
    ...
    ROUTING_PROTOCOL_AODV6,
    ROUTING_PROTOCOL_DYMO,
    ROUTING_PROTOCOL_DYMO6,
    MULTICAST_PROTOCOL_MYPROTOCOL
} NetworkRoutingProtocolType;
```

FIGURE 4-89. Adding MYPROTOCOL to List of Network Layer Protocols

Note **Always add to the end of lists in header files.**

Similarly, add an entry `TRACE_MYPROTOCOL` just before the entry `TRACE_ANY_PROTOCOL` in the enumeration `TraceProtocolType`, defined in the file EXATA_HOME/include/trace.h, as shown in Figure 4-68.

4.4.6.3 Defining Data Structures

Each routing protocol has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Protocol parameters (see Section 4.4.5.5.3.1)
2. Protocol state (see Section 4.4.6.4.3.2)
3. Statistics variables (see Section 4.4.5.10.1)
4. Forwarding table (see Section 4.4.6.4.3.2)

Define an appropriate data structure for MYPROTOCOL called `MyprotocolData` in the protocol header file, multicast_myprotocol.h. As an example, the following data structure (defined in multicast_pim.h) is used by the PIM protocol:

```
typedef struct struct_routing_pim_str
{
    RoutingPimInterface*        interface;
    RoutingPimStats             stats;
    BOOL                        showStat;
    BOOL                        statPrinted;
    RoutingPimModeType          modeType;
    void*                       pimModePtr;
    RandomSeed                  seed;
} PimData;
```

`PimData` stores information for both modes of PIM. In addition, if PIM is operating in dense mode, the following data structure is also used:

```
typedef struct struct_routing_pim_dm_str
{
    RoutingPimDmStats               stats;
    RoutingPimDmForwardingTable     fwdTable;
} PimDmData;
```

In the above declaration, `RoutingPimDmForwardingTable` is the data structure for the PIM-DM forwarding table and `RoutingPimDmStats` is the data structure for PIM-DM statistics.

4.4.6.4 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a Network Layer multicast routing protocol.

4.4.6.4.1 Determining the Protocol Configuration Format

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.5.1).

4.4.6.4.2 Calling the Protocol Initialization Function

The protocol stack of each node is initialized in a bottom-up manner. The initialization of the Network Layer occurs after the layers below it have been initialized. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the Network Layer initialization function NETWORK_Initialize, which is implemented in the file EXATA_HOME/main/network.cpp. Function NETWORK_Initialize, in turn, calls the IP initialization function NetworkIpInit and the routing initialization function IpRoutingInit, which are implemented in the file EXATA_HOME/libraries/developer/src/network_ip.cpp. Function NetworkIpInit in turn calls function NetworkIpParseAndSetRoutingProtocolType, which reads the name of the multicast protocol for each interface from the configuration file and updates the multicast protocol information for that interface. Function IpRoutingInit calls the initialization function of the routing protocol configured on the interface. The code segments from NetworkIpParseAndSetRoutingProtocolType and IpRoutingInit corresponding to PIM are shown in Figure 4-91 and Figure 4-91, respectively. The functions used in the example are explained below.

- Function IO_ReadString reads the name of the routing protocol from the configuration file. The prototype for IO_ReadString is defined in EXATA_HOME/include/fileio.h.

- Function NetworkIpGetInterfaceAddress, defined in network_ip.cpp, returns the IP address associated with an interface.
- Function NetworkIpAddMulticastRoutingProtocolType, defined in network_ip.cpp, initializes the multicast routing protocol information for an interface. In the example of Figure 4-90, NetworkIpAddMulticastRoutingProtocolType updates the `IpInterfaceInfoType` structure associated with the interface (see Section 4.4.3) by setting the `multicastProtocolType` field to `MULTICAST_PROTOCOL_PIM` and the `multicastRoutingProtocol` field to `NULL`.
- Function NetworkIpGetMulticastRoutingProtocol, defined in network_ip.cpp, returns a pointer to the data structure associated with the specified multicast routing protocol. If the same multicast routing protocol is running at multiple interfaces of a node, a single instance of the data structure for the multicast routing protocol is shared by all interfaces. If the multicast routing protocol is running at one of the interfaces and that interface has been assigned a multicast routing protocol structure, NetworkIpGetMulticastRoutingProtocol returns a pointer to that structure; otherwise, it returns `NULL`. In the example of Figure 4-91, NetworkIpGetMulticastRoutingProtocol returns a pointer to the structure `PimData` or `NULL`.
- RoutingPimInit is called if function NetworkIpGetMulticastRoutingProtocol returns `NULL`, i.e., an instance of `PimData` is not associated with any interface. Function RoutingPimInit, defined in multicast_pim.cpp, contains code to initialize PIM. In addition to performing other initializing tasks (see Section 4.4.5.5.3), RoutingPimInit creates an instance of the PIM data structure, `PimData`, and associates it with the specified interface by updating the `multicastRoutingProtocol` field of the `IpInterfaceInfoType` structure associated with the interface to point to the newly created instance of `PimData`.
- Function NetworkIpUpdateMulticastRoutingProtocolAndRouterFunction, defined in network_ip.cpp, is called if function NetworkIpGetMulticastRoutingProtocol returns a non-`NULL` pointer, i.e., if PIM is running at another interface and an instance of `PimData` has been associated with that interface. NetworkIpUpdateMulticastRoutingProtocolAndRouterFunction associates the same instance of `PimData` with the specified interface. This ensures that even if a multicast routing protocol is running at multiple interfaces of a node, all interfaces running the same multicast routing protocol share one instance of the protocol data structure.
- Function IgmpSetMulticastProtocolInfo, defined in EXATA_HOME/libraries/developer/src/multicast_igmp.cpp, is an IGMP function that registers the multicast protocol's function to manage groups. IgmpSetMulticastProtocolInfo is called if IGMP is enabled. A pointer to the PIM function, RoutingPimLocalMembersJoinOrLeave, is passed as a parameter to IgmpSetMulticastProtocolInfo. RoutingPimLocalMembersJoinOrLeave, defined in multicast_pim.cpp, calls function RoutingPimDmLocalMembersJoinOrLeave if PIM is operating in dense mode. RoutingPimDmLocalMembersJoinOrLeave, defined in multicast_pim_dm.cpp, takes appropriate actions when a node joins or leaves a group.

```
void
NetworkIpParseAndSetRoutingProtocolType (Node *node,
                                         const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    BOOL retVal;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
       ...
        IO_ReadString(
                node->nodeId,
                NetworkIpGetInterfaceAddress(node, i),
                nodeInput,
                "MULTICAST-PROTOCOL",
                &retVal,
                protocolString);

              if (retVal)
        {
            ...
            else if (strcmp(protocolString, "PIM") == 0)
            {
                    multicastProtocolType = MULTICAST_PROTOCOL_PIM;
            }
            ...
        }

        NetworkIpAddMulticastRoutingProtocolType(
                        node,
                        multicastProtocolType,
                        i);
         }
     }
     ...
}
```

FIGURE 4-90. Initializing Multicast Routing Protocol Information for an Interface

```
void
IpRoutingInit(Node *node, const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    BOOL retVal;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
      ...
      switch (ip->interfaceInfo[i]->multicastProtocolType)
      {
          case MULTICAST_PROTOCOL_PIM:
          {
              if (!NetworkIpGetMulticastRoutingProtocol(node,
                                        MULTICAST_PROTOCOL_PIM))
              {
                     RoutingPimInit(node, nodeInput, i);
              }
               else
               {
                  NetworkIpUpdateMulticastRoutingProtocolAndRouterFunction(
                              node,
                              MULTICAST_PROTOCOL_PIM,
                              i);

                 /* Inform IGMP about multicast routing protocol */
                 if (ip->isIgmpEnable == TRUE)
                 {
                     IgmpSetMulticastProtocolInfo(
                                   node,
                                   i,
                                   &RoutingPimLocalMembersJoinOrLeave);
                 }
              }
              break;
            } //end case
            ...
        } // end switch
        ...
     } // end for
     ...
}
```

FIGURE 4-91. Calling PIM Initialization Function from IP Initialization Function

Figure 4-92 shows the modifications to be made to IpRoutingInit to incorporate MYPROTOCOL in EXata. RoutingMyprotocolInit is the initialization function for MYPROTOCOL (see Section 4.4.6.4.3) and RoutingMyprotocolLocalMembersJoinOrLeave is the MYPROTOCOL function to handle members joining or leaving groups.

```
void
IpRoutingInit(Node *node, const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    BOOL retVal;
    ...
    for (i = 0; i < node->numberInterfaces; i++)
    {
      ...
             switch (ip->interfaceInfo[i]->multicastProtocolType)
      {
            case MULTICAST_PROTOCOL_PIM:
            {
              ...
            } //end case
            ...
           case MULTICAST_PROTOCOL_MYPROTOCOL:
           {
               if (!NetworkIpGetMulticastRoutingProtocol(
                                          node,
                                          MULTICAST_PROTOCOL_MYPROTOCOL))
               {
                   RoutingMyprotocolInit(node, nodeInput, i);
               }
               else
               {
                   NetworkIpUpdateMulticastRoutingProtocolAndRouterFunction(
                                       node,
                                       MULTICAST_PROTOCOL_MYPROTOCOL,
                                       i);

                    /* Inform IGMP about multicast routing protocol */
                    if (ip->isIgmpEnable == TRUE)
                    {
                        IgmpSetMulticastProtocolInfo(
                                    node,
                                    i,
                                     &RoutingMyprotocolLocalMembersJoinOrLeave);
                    }
                }
                break;
            } //end case
        ...
      }
   }
     ...
}
```

FIGURE 4-92. Calling MYPROTOCOL Initialization Function from IP Initialization Function

4.4.6.4.3 Implementing the Protocol Initialization Function

The initialization of a Network Layer routing protocol takes place in the initialization function of the protocol that is called by the routing initialization function IpRoutingInit (see Figure 4-91). The initialization function of a multicast routing protocol commonly performs the following tasks:

- Create an instance of the protocol data structure
- Read and store the user-specified configuration parameters
- Initialize the state variables, groups, and forwarding table
- Register the protocol's router function and other callback functions with IP and IGMP
- Schedule a timer to itself for starting the protocol

Like all other functions belonging to the protocol, the prototype for the initialization function, RoutingMyprotocolInit, should be included in the protocol's header file, multicast_myprotocol.h.

4.4.6.4.3.1 Creating an Instance and Reading Configuration Parameters

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as flags, connection information, forwarding table used by the protocol, etc.

To store the state, declare the structure to hold the protocol state in the header file, multicast_myprotocol.h (see Section 4.4.5.4). As an example, see the declaration of the PIM-DM data structures `PimData` and `PimDataDm` in multicast_pim.h.

Create an instance of the protocol state by allocating memory to the state structure. PIM-DM performs this task in its initialization function RoutingPimInit by calling the function MEM_malloc to allocate memory for the PIM-DM data structures `PimData` and `PimDmData`, as shown in Figure 4-93. RoutingPimInit and the other PIM-DM functions are implemented in multicast_pim.cpp and multicast_pim_dm.cpp. Data structure and constant definitions for PIM-DM are contained in multicast_pim.h. RoutingPimInit calls the IP function NetworkIpSetMulticastRoutingProtocol, defined in network_ip.cpp, which assigns the newly created `PimData` as the multicast data structure for the specified interface (see Section 4.4.3).

If MYPROTOCOL has any user-specified configuration parameters, these should be read in the protocol's initialization function. PIM-DM does not have any user-specified configuration parameters. To understand how configuration parameters are read from an input file, refer to the AODV example in Section 4.4.5.5.3.1.

```
void RoutingPimInit(Node *node,
                    const NodeInput *nodeInput,
                    int interfaceIndex)
{
    /* Allocate PIM layer structure */
    PimData *pim = (PimData *)
            MEM_malloc (sizeof(PimData));
    PimDmData* pimDmData;
     ...
    /* Determine PIM routing mode */
    IO_ReadString(node->nodeId, interfaceAddress, nodeInput,
                  "PIM-ROUTING-MODE", &retVal, buf);
    ...
    if (strcmp(buf, "DENSE") == 0)
    {
        pim->modeType = ROUTING_PIM_MODE_DENSE;
        pimDmData = (PimDmData*)
                    MEM_malloc (sizeof(PimDmData));

        pim->pimModePtr = (void*) pimDmData;
    }
    ...
    if (ip->ipForwardingEnabled == TRUE)
    {
        NetworkIpAddToMulticastGroupList(node, ALL_PIM_ROUTER);
    }
    /* Set Multicast Routing Protocol */
    NetworkIpSetMulticastRoutingProtocol(node, pim, interfaceIndex);
    /* Inform IGMP about multicast routing protocol */
    if (ip->isIgmpEnable == TRUE)
    {
        IgmpSetMulticastProtocolInfo(node, interfaceIndex,
                &RoutingPimLocalMembersJoinOrLeave);
    }
    if (pim->modeType == ROUTING_PIM_MODE_DENSE)
    {
        /* Set Router function */
        NetworkIpSetMulticastRouterFunction(node,
            &RoutingPimDmRouterFunction, interfaceIndex);
        /* Set funtion pointer to get informed when route changed */
        NetworkIpSetRouteUpdateEventFunction(node,
                &RoutingPimDmAdaptUnicastRouteChange);
         /* Initialize forwarding table */
        RoutingPimDmInitForwardingTable(node);
        ...
    }
    ...
    /* Initializes interface structure */
    RoutingPimInitInterface(node);
    ...
 }
```

FIGURE 4-93. PIM Initialization Function

4.4.6.4.3.2 Initializing State Variables, Groups, and Forwarding Table

The state variables of the routing protocol should be initialized in the protocol's initialization function.

The initialization function of a multicast routing protocol also initializes the list of multicast groups used by the protocol. For PIM, this is done in RoutingPimInit by calling the IP function NetworkIpAddToMulticastGroupList, defined in network_ip.cpp.

The initialization function of a multicast routing protocol also initializes the forwarding table used by the protocol. For PIM-DM, this is done by calling the PIM-DM function RoutingPimDmInitForwardingTable, which is shown in Figure 4-94.

```
void RoutingPimDmInitForwardingTable(Node* node)
{
    PimData* pim = (PimData*)
        NetworkIpGetMulticastRoutingProtocol(node, MULTICAST_PROTOCOL_PIM);
    PimDmData* pimDmData = (PimDmData*)pim->pimModePtr;

    RoutingPimDmForwardingTable* fwdTable = &pimDmData->fwdTable;
    int size =
        sizeof(RoutingPimDmForwardingTableRow)*  PIM_INITIAL_TABLE_SIZE;

    BUFFER_InitializeDataBuffer(&fwdTable->buffer, size);
    fwdTable->numEntries = 0;
}
```

FIGURE 4-94. Initializing PIM-DM Forwarding Table

A routing protocol may need to maintain certain information about its interfaces. In this case, the interface information should be initialized in the initialization function of the protocol. For PIM-DM, this is done in RoutingPimInit by calling the PIM function RoutingPimInitInterface.

4.4.6.4.3.3 Registering Callback Functions with IP and IGMP

Just as a unicast routing protocol, a multicast routing protocol also implements callback functions that it registers with IP during the protocol's initialization. These callback functions are described in Section 4.4.5.5.3.3. In addition, a multicast routing protocol may implement the following callback function that it registers with IP during initialization:

Callback Function: Multicast router function used by the protocol

API to Register Function: NetworkIpSetMulticastRouterFunction

Function Type: MulticastRouterFunctionType

In addition, a multicast protocol also interacts with the IGMP protocol and implements the following callback function that it registers with IGMP during the protocol's initialization. See EXATA_HOME/libraries/developer/src/multicast_igmp.cpp for a description of the function parameters.

Callback Function: Function used by the protocol to handle members leaving or joining groups

API to Register Function: IgmpSetMulticastProtocolInfo

Function Type: MulticastProtocolType

As an example, the PIM-DM initialization function RoutingPimInit (see Figure 4-93) calls function NetworkIpSetMulticastRouterFunction to register with IP the PIM-DM router function, RoutingPimDmRouterFunction. RoutingPimInit calls function NetworkIpSetSetRouteUpdateEventFunction to register with IP the PIM-DM function, RoutingPimDmAdaptUnicastRouteChange, which handles

changes in the unicast routes. If IGMP is enabled, RoutingPimInit calls function IgmpSetMulticastProtocolInfo to register with IGMP the PIM function, RoutingPimLocalMembersJoinOrleave, which handles members joining or leaving multicast groups. RoutingPimLocalMembersJoinOrLeave calls function RoutingPimDmLocalMembersJoinOrLeave if PIM is operating in dense mode. RoutingPimDmLocalMembersJoinOrLeave, defined in multicast_pim_dm.cpp, takes appropriate actions when a node joins or leaves a group.

Note that function IgmpSetMulticastProtocolInfo is called by function NetworkIpInit (see Section 4.4.6.4.2) as well as function RoutingPimInit. This is because function RoutingPimLocalMembersJoinOrleave has to be registered for each interface on which PIM is running. The other callback functions are registered only once for a node.

4.4.6.4.3.4 Initializing Timers

A routing protocol may need to set a timer at initialization. For example, the PIM initialization function RoutingPimInit sets a timer of type `MSG_ROUTING_PimScheduleHello` to trigger after a random delay. See Section 3.3.2.2 for details on setting timers.

4.4.6.5 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a Network Layer multicast routing protocol. Some steps are the same as for adding a unicast routing protocol, described in Section 4.4.5.6.

4.4.6.5.1 Modifying the IP Event Dispatcher

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.6.1). To enable the protocol MYPROTOCOL to receive events, add a case in the switch statement in the IP event dispatcher function, NetworkIpLayer, to call MYPROTOCOL's event dispatcher function, MyprotocolHandleProtocolEvent, when the protocol type of the received message is `MULTICAST_PROTOCOL_MYPROTOCOL`.

4.4.6.5.2 Implementing the Protocol Event Dispatcher

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.6.2).

Declare any additional event types used by the protocol in the enumeration file api.h, as shown in Figure 4-77.

Write the event dispatcher for MYPROTOCOL, MyprotocolHandleProtocolEvent, which should include a switch on all message types that the protocol may receive. The event dispatcher function for a routing protocol generally handles timer events. As an example, Figure 4-95 shows the PIM event dispatcher function RoutingPimHandleProtocolEvent.

A multicast routing protocol typically performs the following functions which are triggered by time outs:

- Send *hello* packets and other routing packets periodically. For example, function RoutingPimHandleProtocolEvent calls function RoutingPimSendHelloPacket to send *hello* packets when the timer event `MSG_ROUTING_PimScheduleHello` occurs.
- Perform pruning functions to handle members leaving a multicast group. For example, RoutingPimHandleProtocolEvent performs pruning operations when timer event `MSG_ROUTING_PimPruneTimeoutAlarm` occurs.
- Perform grafting functions to handle members joining a multicast group. For example, RoutingPimHandleProtocolEvent performs grafting operations when timer event `MSG_ROUTING_PimGraftRtmxtTimeOut` occurs.

As part of handling an event, a routing protocol at a node may transmit packets to its peers. See Section 4.4.5.6.2 for details of sending packets.

```
void RoutingPimHandleProtocolEvent(Node *node, Message *msg)
{
    ...
    switch (msg->eventType)
    {
        case MSG_ROUTING_PimScheduleHello:
        {
            ...
            /* Send Hello packet on all of its interfaces */
            RoutingPimSendHelloPacket(node, interfaceIndex);
            /* Reschedule Hello packet broadcast */
            newMsg = MESSAGE_Alloc(node,
                                   NETWORK_LAYER,
                                   MULTICAST_PROTOCOL_PIM,
                                   MSG_ROUTING_PimScheduleHello);
            MESSAGE_AddInfo(node, newMsg, sizeof(int), INFO_TYPE_PhyIndex);
            memcpy(MESSAGE_ReturnInfo(newMsg, INFO_TYPE_PhyIndex),
                   &interfaceIndex, sizeof(int));
            MESSAGE_Send(node, newMsg,
                         pim->interface[interfaceIndex].helloInterval);
            break;
        }
        ...
        case MSG_ROUTING_PimDmPruneTimeoutAlarm:
        {
           ...
        }
        ...
        case MSG_ROUTING_PimDmGraftRtmxtTimeOut:
        {
           ...
        }
        case MSG_ROUTING_PimDmJoinTimeOut:
        {
           ...
        }
        case MSG_ROUTING_PimDmScheduleJoin:
        {
           ...
        }
        ...
        default:
        {
            printf("    Event Type = %d\n", msg->eventType);
            ERROR_Assert(FALSE, "Unknown protocol event in PIM\n");
        }
    }
    MESSAGE_Free(node, msg);
}
```

FIGURE 4-95. PIM Event Dispatcher

4.4.6.6 Processing Routing Packets

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.8).

4.4.6.6.1 Modifying IP Packet Handler

To add a new multicast routing protocol, MYPROTOCOL, at the Network Layer, assign an IP protocol number to MYPROTOCOL and modify the IP function DeliverPacket to deliver packets to MYPROTOCOL.

1. Define an IP Protocol Number for MYPROTOCOL, `IPPROTOCOL_MYPROTOCOL`, in the file network_ip.h. See Section 4.4.5.8.1 for details.
2. Modify the IP function DeliverPacket, defined in network_ip.cpp, to deliver packets to MYPROTOCOL.
   This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.8.1).

   Add code to DeliverPacket to call MYPROTOCOL's packet handler function, RoutingMyprotocolHandleProtocolPacket, with appropriate parameters, if the IP Protocol Number read from the packet's header is `IPPROTO_MYPROTOCOL` (defined in step 1) as shown in Figure 4-96.

```
static void //inline//
DeliverPacket(Node *node, Message *msg,
              int interfaceIndex, NodeAddress previousHopAddress)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    NodeAddress sourceAddress = 0;
    NodeAddress destinationAddress =0;
    unsigned char ipProtocolNumber;
    unsigned ttl =0;
    ...
    switch (ipProtocolNumber)
    {
        ...
        // Delivery to network-layer routing protocols.
        ...
        case IPPROTO_PIM:
        {
            RoutingPimHandleProtocolPacket(node, msg, sourceAddress,
                                           interfaceIndex);
            break;
        }
        case IPPROTO_MYPROTOCOL:
        {
            RoutingMyprotocolHandleProtocolPacket(node, msg, ...);
            break;
        }
        ...
    }//switch//
}//DeliverPacket//
```

FIGURE 4-96. Delivering Packets from IP to Network Layer Routing Protocols

4.4.6.6.2 Implementing the Protocol Packet Handler

A routing protocol's packet handler should include a switch on all types of packets that the protocol may receive. It can then process each packet type either inside the switch or by calling a function to handle the packet type received. For example, function RoutingPimHandleProtocolPacket, shown in Figure 4-97 and implemented in multicast_pim.cpp, processes PIM routing packets.

As part of processing a received packet, a routing protocol at a node may transmit packets to its peers. See Section 4.4.5.6.2 for details of sending packets.

Write function RoutingMyprotocolHandleProtocolPacket to handle routing packets for MYPROTOCOL. Follow the example of RoutingPimHandleProtocolPacket or the packet handler for some other Network Layer multicast routing protocol in EXata.

```
void RoutingPimHandleProtocolPacket(Node *node, Message *msg,
                                    NodeAddress srcAddr, int interfaceId)
{
    PimData* pim = (PimData*)
        NetworkIpGetMulticastRoutingProtocol(node, MULTICAST_PROTOCOL_PIM);

    /* Get PIM Common header */
    RoutingPimCommonHeaderType *commonHeader =
        (RoutingPimCommonHeaderType *) MESSAGE_ReturnPacket(msg);

    /* Make sure that PIM is running over this interface */
    if (!RoutingPimIsPimEnabledInterface(node, interfaceId))
    {
        MESSAGE_Free(node, msg);
        return;
    }
    switch (RoutingPimCommonHeaderGetType(commonHeader->rpChType))    {
        case ROUTING_PIM_HELLO:
        {
            RoutingPimHelloPacket *helloPkt =
                (RoutingPimHelloPacket *) MESSAGE_ReturnPacket(msg);
            int size = MESSAGE_ReturnPacketSize(msg);
            ...
             if (pim->modeType == ROUTING_PIM_MODE_DENSE)
            {
                PimDmData* pimDmData = (PimDmData*) pim->pimModePtr;
                pimDmData->stats.helloReceived++;
                RoutingPimDmHandleHelloPacket(node, srcAddr, helloPkt, size,
                                              interfaceId);
            }
            else
            {
                ...
            }
            break;
        }
        case ROUTING_PIM_JOIN_PRUNE:
        {
            ...
        }
        ...
        default:
        {
            ERROR_Assert(FALSE, "Unknown packet type\n");
        }
    }
    MESSAGE_Free(node, msg);
}
```

FIGURE 4-97. PIM Routing Packet Handling Function

4.4.6.7 Implementing Callback Functions

As explained in Section 4.4.6.4.3.3, a Network Layer multicast routing protocol in EXata implements certain callback functions. The implementation of these functions is similar to those for a unicast routing protocol (see Section 4.4.5.9).

If the multicast protocol uses IGMP services, it should implement a callback function that handles members joining or leaving a group. As an example, function RoutingPimDmLocalMembersJoinorLeave, shown in Figure 4-98 and implemented in multicast_pim_dm.cpp, is the PIM-DM function that takes appropriate actions when a member joins or leaves a group. This function is called by function RoutingPimLocalMembersJoinorLeave, which is registered with IGMP during the protocol's initialization (see Section 4.4.6.4.3.3).

```
void RoutingPimDmLocalMembersJoinOrLeave(Node *node,
                                         NodeAddress groupAddr,
                                         int interfaceId,
                                         LocalGroupMembershipEventType event)
{
    PimData* pim = (PimData*)
        NetworkIpGetMulticastRoutingProtocol(node, MULTICAST_PROTOCOL_PIM);
    PimDmData* pimDmData = (PimDmData*)pim->pimModePtr;

    RoutingPimDmForwardingTableRow* rowPtr;
    RoutingPimDmDownstreamListItem* downstreamInfo;
    RoutingPimInterface* thisInterface;
    unsigned int i;

    rowPtr = (RoutingPimDmForwardingTableRow *)
                    BUFFER_GetData(&pimDmData->fwdTable.buffer);
    thisInterface = &pim->interface[interfaceId];

    switch (event)
    {
        case LOCAL_MEMBER_JOIN_GROUP:
        {
            ...
        }

        case LOCAL_MEMBER_LEAVE_GROUP:
        {
            ...
        }

        default:
        {
            ERROR_Assert(FALSE, "Unknown IGMP Event\n");
        }
    }
}
```

FIGURE 4-98. PIM-DM Function to Handle Members Leaving or Joining a Group

Write the callback functions that MYPROTOCOL may need. Use PIM or some other appropriate multicast routing protocol as an example. Register all callback functions with IP and IGMP in the protocol's initialization function (see Section 4.4.6.4.3.3).

4.4.6.8 Collecting and Reporting Statistics

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.10).

4.4.6.9 Finalization

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.11).

4.4.6.10 Including and Compiling Files

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.12).

4.4.6.11 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.4.7 EXata Queuing Protocols

This section describes the queuing protocols implemented in EXata. Section 4.4.7.1 and Section 4.4.7.2 describe the implementation of queues in EXata. Section 4.4.7.3 describes how EXata protocol models use the existing queuing models. Section 4.4.7.4 describes the procedure to add a new queue model to EXata.

4.4.7.1 Data Structures and Classes

EXata implements several queue management algorithms (see Table 4-10). Queues are implemented using C++ classes. This section gives details of the base class that implements the FIFO queue and from which classes that implement other queuing disciplines are derived. This section also describes some of the data structures used in the implementation of queues. These data structures and classes are defined in EXATA_HOME/include/if_queue.h. (Note that only a partial description of the data structures is provided here. Refer to the file if_queue.h for a complete description.)

1. `QueueOperation`**:** This enumeration type lists the different types of dequeue operations.

```
typedef enum
{
    PEEK_AT_NEXT_PACKET, // 0 : Handles false dequeue functionality
    DEQUEUE_PACKET,      // 1 : Handles dequeue functionality
    DISCARD_PACKET,      // 2 : Handles drop functionality
    DROP_PACKET          // 3 : Handles forcefully drop functionality
} QueueOperation;
```

2. `PacketArrayEntry`: This data structure represents an entry in the array of stored messages. The fields of the data structure are described below.

```
typedef struct packet_array_entry_str
{
    Message *msg;
    clocktype insertTime;
    double infoField[PACKET_ARRAY_INFO_FIELD_SIZE / sizeof(double)];
    double serviceTag;
} PacketArrayEntry;
```

   - `msg`: A pointer to the message.
   - `insertTime`: Simulation time when the message is inserted in the queue.
   - `infoField`: Array that stores queuing algorithm-dependent data.
   - `serviceTag`: Variable that stores user-specific data.

3. `Queue`: This is the base class that is used to derive specific queue classes. This class implements the FIFO discipline and all other queue disciplines are derived from it. Figure 4-99, Figure 4-100 and Figure 4-101 show the declaration of the `Queue` class.

```
class Queue
{
  protected:
    PacketArrayEntry* packetArray;
    int numPackets;
    int maxPackets;
    int infoFieldSize;
    int bytesUsed;
    int queueSizeInBytes;
    int headIndex;
    int tailIndex;
    ...
    // QOS interface observation statistics
    int qDelay;
    int totalTransmission;
    clocktype qosMonitorInterval;
    clocktype queueCreationTime;

    // currentPeriod statistics
    clocktype currentStateStartTime;
    clocktype utilizedTime;
    ...
    clocktype currentPeriodStartTime;
    ...
    // standard statistics collected
    float delayAveragingWeight; // used to calculate running average delay
    BOOL isCollectStats;
    D_Float64 avgSize;
    int peakSize;
    int numPacketsQueued;
    int numPacketsDequeued;
    D_Float64 numPacketsDropped;
     ...
    clocktype lastChange;
    clocktype totalDelays;
    clocktype longestDelay;

    // Utility functions
    inline int  RetriveArrayIndex(int index);
    void UpdateQueueLengthStats(const clocktype currentTime);
    void UpdateQueueDelayStats(int packetArrayIndex,
                               const clocktype currentTime);
    void FinalizeQueue(Node *node,
                       const char *layer,
                       const char *protocol,
                       const int interfaceIndex,
                       const int instanceId,
                       const char *invokingProtocol);
 public:
   ...
};
```

FIGURE 4-99. Declaration of Class `Queue`: Protected Members

```
class Queue
{
  protected:
    ...
  public:

    Queue(){};
    ~Queue();

    virtual void SetupQueue(Node* node,
                            const char queueTypeString[],
                            const int queueSize,
                            const int infoFieldSize = 0,
                            const int interfaceIndex = 0,
                            const int queueNumber = 0,
                            const BOOL enableQueueStat = FALSE,
                            const BOOL showQueueInGui = FALSE,
                            const clocktype currentTime = 0,
                            const void* configInfo = NULL);

    virtual void insert(Message* msg,
                        const void* infoField,
                        BOOL* QueueIsFull,
                        const clocktype currentTime,
                        const double serviceTag = 0.0);

    virtual void insert(Message* msg,
                        const void* infoField,
                        BOOL* QueueIsFull,
                        const clocktype currentTime,
                        TosType* tos,
                        const double serviceTag = 0.0);

    virtual BOOL retrieve(Message** msg,
                          const int index,
                          const QueueOperation operation,
                          const clocktype currentTime,
                          double* serviceTag = NULL);

    virtual BOOL isEmpty();

    virtual int bytesInQueue();

    virtual int freeSpaceInQueue();

    virtual int packetsInQueue();
    ...
}
```

FIGURE 4-100. Declaration of Class Queue: Public Members (Part 1)

```
class Queue
{
  protected:
    ...
  public:

    Queue(){};
    ~Queue();
    ...
    virtual int packetsInQueue();

    inline int sizeOfQueue();

    void setServiceTag(double serviceTag);
    virtual int replicate(Queue* newQueue);

    // Resource Management API
    void setQueueBehavior(BOOL suspend = FALSE);

    // QOS interface observation API
    virtual void qosQueueInformationUpdate(int* qDelayVal,
                                int* totalTransmissionVal,
                                const clocktype currentTime,
                                BOOL isResetTotalTransmissionVal = FALSE);

    // CurrentPeriod statistics API
    inline int byteDequeuedInPeriod();

    inline clocktype utilizationInPeriod();

    inline clocktype averageTimeInQueue();

    inline void resetPeriod (clocktype currentTime);

    inline clocktype periodStartTime();


    virtual void finalize(Node* node,
                          const char* layer,
                          const int interfaceIndex,
                          const int instanceId,
                          const char* invokingProtocol = "IP",
                          const char* splStatStr = NULL);

    clocktype getPacketInsertTime(int pktIndex);

};
```

FIGURE 4-101. Declaration of Class Queue: Public Members (Part 2)

Some of the data members of the Queue class are described below. The utility functions of the Queue class are described in Section 4.4.7.2.

- packetArray: Array of packets in the queue.
- numPackets: Number of packets in the queue.
- maxPackets: Maximum number of packets that the queue can hold.
- infoFieldSize: Size of the infoField array in PacketArrayEntry.
- bytesUsed: Total number of bytes in the packets in the queue.
- queueSizeInBytes: Maximum queue size, in bytes.
- headIndex: Index of the packet at the head of the queue.
- tailIndex: Index of the packet at the tail of the queue.
- qDelay: Average packet delay in the current measurement period. A measurement period is the time period from the last call to function resetPeriod (or from the beginning of simulation, if resetPeriod has not been called) to the current simulation time.
- totalTransmission: Total number of bytes removed from the queue in the current measurement period.
- queueCreationTime: Simulation time when the queue was created.
- currentPeriodStartTime: Start time of the current measurement period.
- utilizedTime: Total time that the queue was non-empty in the current measurement period.
- delayAveragingWeight: Weight used to calculate the running average delay.
- avgSize: Average queue size, in bytes, since the start of simulation.
- peakSize: Largest queue size, in bytes, since the start of simulation.
- numPacketsQueued: Number of packets enqueued since the start of simulation.
- numPacketsDequeued: Number of packets dequeued since the start of simulation.
- numPacketsDropped: Number of packets dropped since the start of simulation.
- numBytesQueued: Number of bytes enqueued since the start of simulation.
- lastChange: Simulation time when the statistics were last updated.
- totalDelays: Total of the packet delays since the start of simulation.
- longestDelay: Longest packet delay since the start of simulation.

4.4.7.2 Interface Functions

The protected interface functions provided by the Queue class are listed below. These functions are implemented in EXATA_HOME/libraries/developer/src/if_queue.cpp.

- RetriveArrayIndex: This function returns the position of a packet in the queue.
- UpdateQueueLengthStats: This function calculates the average queue size.
- UpdateQueueDelayStats: This function calculates the queue delay.
- FinalizeQueue: This is the finalization function for the queue and prints the queue statistics.

In addition to the functions listed above, the Queue class contains prototypes for the following virtual functions. See file if_queue.cpp for an explanation of the parameters of these functions as well as the implementations of the functions for the base class. A class derived from the Queue class can provide alternate implementation of these functions.

- SetupQueue: This function initializes a queue.
- insert: This function inserts a message into a queue.

- retrieve: This function dequeues, drops, or takes a peek at a message in a queue.
- isEmpty: This function determines if a queue is empty.
- bytesInQueue: This function returns the total number of bytes stored in a queue.
- freeSpaceInQueue: This function returns the total number of free bytes available in the buffer.
- packetsInQueue: This function returns the total number of packets stored in a queue.
- sizeOfQueue: This function returns the maximum size of a queue, in bytes.
- setServiceTag: This function updates the service tag of the last enqueued packet.
- replicate: This function replicates a queue.
- qosQueueInformationUpdate: This function updates QOS information.
- byteDequeuedInPeriod: This function returns the total number of bytes dequeued from a queue in a the current measurement period, i.e., the time period from the last call to function resetPeriod (or from the beginning of simulation, if resetPeriod has not been called) to the current simulation time.
- utilizationInPeriod: This function returns the queue utilization, i.e., the amount of time that the queue is non-empty, in the current measurement period.
- averageTimeInQueue: This function returns the average time a packet spends in a queue in the current measurement period.
- resetPeriod: This function resets the measurement period and the period statistics variables.
- periodStartTime: This function returns the start time of the current measurement period.
- finalize: This function outputs the final queue statistics by calling FinalizeQueue.
- get PacketInsertTime: This function returns the insertion time of the top packet of the queue. If the queue is empty, it returns 0.

In addition to the interface functions provided by the `Queue` class, function QUEUE_Setup is also available for implementing queue management algorithms. Function QUEUE_Setup is used to create and initialize an object of the base `Queue` class or an object of a class derived from the base `Queue` class. The prototype of QUEUE_Setup is contained in if_queue.h and the implementation is contained in if_queue.cpp. The prototype of QUEUE_Setup is shown in Figure 4-102. See file if_queue.h for a description of the function parameters.

```
void QUEUE_Setup(
    Node* node,
    Queue** queue,
    const char queueTypeString[],
    const int queueSize,
    const int interfaceIndex,
    const int queueNumber,
    const int infoFieldSize = 0,
    const BOOL enableQueueStat = FALSE,
    const BOOL showQueueInGui = FALSE,
    const clocktype currentTime = 0,
    const void* configInfo = NULL);
```

FIGURE 4-102. Prototype of Function QUEUE_Setup

4.4.7.3 Using the `Queue` Class

This section describes how protocols use the `Queue` class and interface functions to implement queues.

The `Queue` class can be used to implement queues at any layer. In this section, we illustrate the use of queues by taking code segments from the implementation of the Messenger application, which is implemented by files app_messenger.h and app_messenger.cpp in the folder EXATA_HOME/libraries/developer/src.

4.4.7.3.1 Creating and Initializing a Queue

To create a queue, declare a pointer variable that points to an object of the `Queue` class and call function QUEUE_Setup (see Section 4.4.7.2). The type of queue that is created is determined by the parameter `queueTypeString` of QUEUE_Setup. Table 4-12 shows the different types of queues implemented in EXata. Other queue types can be derived from the `Queue` class (see Section 4.4.7.4).

TABLE 4-12. Implemented Queue Types

queueTypeString	Queue Type
"FIFO"	FIFO queue
"RED"	Random Early Detection queue
"RED-ECN"	RED with Explicit Congestion Notification
"WRED"	Weighted RED queue
"RIO"	RED with In/Out bit
"ATM-RED"	RED for use in ATM networks

As an example, Figure 4-103 shows the Messenger function MessengerOutputQueueInitialize which initializes a FIFO queue by calling function QUEUE_Setup. MessengerOutputQueueInitialize is implemented in app_messenger.cpp and the data structure `MessengerState` is defined in app_messenger.h.

```
static void
MessengerOutputQueueInitialize(
    Node *node,
    MessengerState *messenger)
{
    Queue* queuePtr = NULL;

    // a single FIFO queue
    QUEUE_Setup(
        node,
        &queuePtr,
        "FIFO",
        DEFAULT_APP_QUEUE_SIZE,
        APP_MESSENGER, // this is used to set the random seed
        0,
        0, // infoFieldSize
        FALSE,
        FALSE,
        getSimTime(node),
        NULL);

    messenger->queue = queuePtr;
}
```

FIGURE 4-103. Creating and Initializing a Queue

4.4.7.3.2 Performing Queue Operations

Queue operations, such as insertion, deletion, and checking if the queue is empty, are performed by calling the interface functions defined for the `Queue` class (see Section 4.4.7.2). The interface functions for the base class, `Queue`, are implemented in if_queue.cpp.

As an example, Figure 4-104 shows the Messenger function MessengerSendAllFromOutputQueue. MessengerSendAllFromOutputQueue calls the `Queue` class interface functions retrieve and packetsInQueue to dequeue packets from a queue. Other interface functions can be called in a similar way to perform queue operations.

```
static void
MessengerSendAllFromOutputQueue(Node *node, MessengerState *messenger)
{
    Queue* queuePtr = messenger->queue;
    int packetIndex = 0;

    // Dequeue all the packets from the queue which are currently there.
    while (queuePtr->packetsInQueue())
    {
        Message *queueMsg = NULL;
        ...
        queuePtr->retrieve(&queueMsg,
                           packetIndex,
                           DEQUEUE_PACKET,
                           TIME_getSimTime(node));
        if (queueMsg != NULL)
        {
            if (DEBUG_QUEUE)
            {
                printf("\t\tDequeuing a packet from Queue\n");
            }
            if (TEST_VOICE_APP)
            {
                ...
            }
            messenger->messageLastSentTime = TIME_getSimTime(node);
            // Send All Queued Messeges
            MESSAGE_Send(node, queueMsg, PROCESS_IMMEDIATELY);
        }
    }
}
```

FIGURE 4-104. Calling Interface Functions for Queue Operations

4.4.7.4 Adding a New Queue Model

This section describes the procedure to add a new queue model to EXata. It describes how to develop code components common to most queuing protocols such as deriving a new queue class from the base class, reading queue-specific parameters, and implementing interface functions.

We illustrate the process of adding a queue model by using as an example the implementation code for the RED (Random Early Detection) queuing protocol. The header file for the RED implementation is queue_red.h and the source file is queue_red.cpp in the folder EXATA_HOME/libraries/developer/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a queuing protocol. After understanding the discussed snippets, look at the complete code for RED to understand how a queue model is derived from the FIFO queue model described in Section 4.4.7.1 and Section 4.4.7.2.

The following list summarizes the actions that need to be performed for adding a new queue model, MYQUEUE, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.4.7.4.1).
2. Modify the file if_queue.cpp to include the queue model's header file (see Section 4.4.7.4.1).
3. Define data structures for the queue model (see Section 4.4.7.4.2).
4. Decide on the format for the queue model-specific configuration parameters (see Section 4.4.7.4.3).
5. Write a function to read the queue model-specific configuration parameters (see Section 4.4.7.4.4).

6. Derive the new queue class based on the base `Queue` class (see Section 4.4.7.4.5).
7. Implement interface functions for the new queue model (see Section 4.4.7.4.6).
8. Modify function QUEUE_Setup to call MYQUEUE's setup function (see Section 4.4.7.4.7)
9. Include the queue model's header and source files in the EXata tree and compile (see Section 4.4.7.4.8).
10. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.4.7.4.9).

4.4.7.4.1 Creating Files

This step is similar to the one for Network Layer routing protocols (see Section 4.4.5.2). Create the header and source files for the queue model. Name these files in a way that clearly indicates the model that they implement. For queue models, prefix the file names with *queue_*.

Examples:

- queue_red.cpp, queue.h: These files in the folder EXATA_HOME/libraries/developer/src implement the RED queueing discipline.
- queue_rio_ecn.cpp, queue_rio_ecn.h: These files in the folder EXATA_HOME/libraries/developer/src implement the RIO ECN queueing discipline.

For the example queue, MYQUEUE, create files queue_myqueue.h and queue_myqueue.cpp in the appropriate folder. If MYQUEUE is a general-purpose queue, add the files to the folder EXATA_HOME/libraries/user_models/src (see Section 4.4.5.2). If MYQUEUE is meant to be used by a specific protocol, add it to the same folder as the protocol files.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems, even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file for the queue model, queue_myqueue.h, should contain the following:

- Constant definitions
- Data structure definitions
- Class definition for the new queue model derived from the base `Queue` class (see Section 4.4.7.1)
- Prototypes for any additional interface functions in the queue model's source file
- Statement to include the generic queue model's header file:

```
#include "if_queue.h"
```

The source file for the queue model, queue_myqueue.cpp, should contain the following:

- Statement to include the queue model's header file:

```
#include "queue_myqueue.h"
```

- Statements to include standard library functions and other header files needed by the queue model's source file. A typical queue model source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "api.h"        // EXATA_HOME/include/api.h
#include "if_queue.h"   // EXATA_HOME/include/if_queue.h. This allows
                        // for the use of the base Queue class.
```

- Function to read MYQUEUE parameters, if any, from the configuration file
- Interface functions for MYQUEUE

The file EXATA_HOME/libraries/developer/src/if_queue.cpp contains the generic function to setup a queue, QUEUE_Setup. QUEUE_Setup in turn calls setup functions for specific queue types. To make the MYQUEUE setup function available to function QUEUE_Setup, insert the following include statement in the file if_queue.cpp:

```
#include "queue_myqueue.h"
```

4.4.7.4.2 Defining Data Structures

In EXata, queue types are defined as classes derived from the base `Queue` class. In addition to the variables that are part of `Queue` class, a queue type may require other variables. For example, the RED queue implementation uses the data structure `RedParameters`, defined in queue_red.h, to store the queue-specific parameters.

```
typedef struct
{
    int             minThreshold;
    int             maxThreshold;
    double          maxProbability;
    double          queueWeight;
    clocktype       typicalSmallPacketTransmissionTime;
} RedParameters;
```

For MYQUEUE, define an appropriate data structure for MYQUEUE-specific parameters, if needed, in the file queue_myqueue.h.

4.4.7.4.3 Determining the Queue Configuration Format

A queue model may use model-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a queue model's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>` : Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.

<Parameter-name> : Name of the parameter.

<Index> : Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.

<Parameter-value> : Value to be used for the parameter.

As an example, the following configuration specifies that the RED queue is to be used and gives the values of the parameters for the RED queue. Refer to file EXATA_HOME/scenarios/default/default.config for an explanation of these parameters.

```
IP-QUEUE-TYPE          RED
RED-MIN-THRESHOLD      5
RED-MAX-THRESHOLD      15
RED-MAX-PROBABILITY    0.02
RED-QUEUE-WEIGHT       0.002
RED-SMALL-PACKET-TRANSMISSION-TIME    10MS
```

A configuration parameter is not always mandatory. If an optional configuration parameter is not assigned a value, the default value is used. For example, if a user does not specify a value for RED-MIN-THRESHOLD, the default value of 5 (DEFAULT_RED_MIN_THRESHOLD) is used by the model.

Decide on the format for specifying the new queue model's configuration parameters. For our example queue, specify the configuration parameters in the EXata configuration file using the following format (<Identifier> and <Index> can also be used to qualify the parameter declarations, as described above):

```
IP-QUEUE-TYPE          MYQUEUE
<param1>               <value1>
...
<paramN>               <valueN>
```

where

<param1>, ..., <paramN> : Names of parameters for MYQUEUE.

<value1>, ..., <valueN> : Values of the queue parameters.

Section 4.4.7.4.4 explains how to read user input specified in this format.

4.4.7.4.4 Reading Configuration Parameters

This section explains how to read user-specified configuration parameters for queue models (see Section 4.4.7.4.3) and provide them to the queue setup function.

As an example, Figure 4-105 shows how IP function NetworkIpInitOutputQueueConfiguration reads the configuration parameters for the RED queue by calling function ReadRedConfigurationParameters and initializes a RED queue by calling the generic queue setup function QUEUE_Setup. NetworkIpInitOutputQueueConfiguration is implemented in network_ip.cpp. ReadRedConfigurationParameters is implemented in queue_red.cpp. QUEUE_Setup is implemented in if_queue.cpp.

```
void
NetworkIpInitOutputQueueConfiguration(
    Node *node,
    const NodeInput *nodeInput,
    int interfaceIndex)
{
    ...
    for (i = 0; i < numPriorities; i++)
    {
        Queue* queue = NULL;
        char queueTypeString[MAX_STRING_LENGTH] = {0};
        ...
        void* spConfigInfo = NULL; // Queue Specific configurations.

        IO_ReadStringInstance(
            node, node->nodeId, interfaceIndex, nodeInput, "IP-QUEUE-TYPE",
            i, TRUE, &wasFound,  queueTypeString);
        ...

        if (!strcmp(queueTypeString, "FIFO"))
        {
            // No specific configuration for FIFO
        }
        else if (!strcmp(queueTypeString, "RED"))
        {
            IO_ReadString(
                node, node->nodeId, interfaceIndex,
                nodeInput, "ECN", &wasFound, buf);
            if (wasFound && (!strcmp(buf, "YES")))
            {
                ...
            }
            else
            {
                RedParameters* redParams = NULL;
               ReadRedConfigurationParameters(node, interfaceIndex, nodeInput,
                                              enableQueueStat, i, &redParams);
                spConfigInfo = (void*)(redParams);
            }
        }
         else
         ...
       // Initialize Queue depending on queueTypeString specification
        QUEUE_Setup(node, &queue, queueTypeString, queueSize, interfaceIndex,
                    priority, 0, enableQueueStat, node->guiOption,
                    getSimTime(node), spConfigInfo);
      ...
    }
}
```

FIGURE 4-105. Setting Up Queues

Figure 4-106 shows the RED queue function ReadRedConfigurationParameters. ReadRedConfigurationParameters uses IO functions such as IO_ReadTimeInstance and IO_ReadIntInstance to read parameter values from the input file and store them in the appropriate fields of the RED queue data structure `RedParameters` (see Section 4.4.7.4.2). If a value is not specified for a parameter in the input file, ReadRedConfigurationParameters stores the default value for that parameter.

IO_ReadTimeInstance, IO_ReadIntInstance, and other IO functions are defined in EXATA_HOME/include/fileio.h.

For the example queue, MYQUEUE, write a function to read user-specified configuration parameters if MYQUEUE uses such parameters.

```
void ReadRedConfigurationParameters(
    Node* node,
    int interfaceIndex,
    const NodeInput* nodeInput,
    BOOL enableQueueStat,
    int queueIndex,
    RedParameters** redConfigParams)
{
    BOOL retVal = FALSE;
    RedParameters* red = NULL;
    int nodeId = node->nodeId;

    red = (RedParameters*) MEM_malloc(sizeof(RedParameters));
    memset(red, 0, sizeof(RedParameters));
     IO_ReadTimeInstance(
        node,
        nodeId,
        interfaceIndex,
        nodeInput,
        "RED-SMALL-PACKET-TRANSMISSION-TIME",
        queueIndex,  // parameterInstanceNumber
        TRUE,      // fallbackIfNoInstanceMatch
        &retVal,
        &(red->typicalSmallPacketTransmissionTime));

    if (!retVal)
    {
        red->typicalSmallPacketTransmissionTime =
                DEFAULT_RED_SMALL_PACKET_TRANSMISSION_TIME;
    }
    ...
    // Attch Info
    *redConfigParams = red;
}
```

FIGURE 4-106. Reading Queue Configuration Parameters

4.4.7.4.5 Deriving New Queue Class from Base `Queue` Class

In EXata all queues are implemented as classes derived from the base `Queue` class (see Section 4.4.7.1 and Section 4.4.7.2). This section describes how to implement a new queue type as a class derived from the base `Queue` class by taking as an example the implementation of the RED queue.

Figure 4-107 shows the declaration of the class `RedQueue` that implements the RED queue model. `RedQueue` is declared in queue_red.h and is derived from the base class `Queue`. One of the variables declared in `RedQueue` is a pointer to the data structure `RedParameters` (see Section 4.4.7.4.2) which stores the RED queue configurable parameters. Class `RedQueue` also contains prototypes of interface functions that are specific to RED queue or that override the base `Queue` class functions. Implementation of these functions is discussed in Section 4.4.7.4.6.

For the example queue, MYQUEUE, declare a class `Myqueue` derived from the base class `Queue` in the file queue_myqueue.h.

```
class RedQueue:public Queue
{
    protected:
      clocktype         startIdleTime; // Start of idle time
      double            averageQueueSize;
      RedParameters*    redParams;
      int               packetCount; // packet since last marked packet
      RandomSeed        randomDropSeed; // A random seed
      // Utility Functions
      void UpdateAverageQueueSize(const BOOL queueIsEmpty,
                                  const int numPackets,
                                  const double queueWeight,
                                  const clocktype smallPktTxTime,
                                  const clocktype startIdleTime,
                                  double* avgQueueSize,
                                  const clocktype theTime);
      BOOL RedDropThePacket();
    public:
      virtual void SetupQueue(Node* node,
                              const char queueTypeString[],
                              const int queueSize,
                              const int interfaceIndex,
                              const int queueNumber,
                              const int infoFieldSize = 0,
                              const BOOL enableQueueStat = FALSE,
                              const BOOL showQueueInGui = FALSE,
                              const clocktype currentTime = 0,
                              const void* configInfo = NULL);
      virtual void insert(Message* msg,
                          const void* infoField,
                          BOOL* QueueIsFull,
                          const clocktype currentTime,
                          const double serviceTag = 0.0);
      virtual void insert(Message* msg,
                          const void* infoField,
                          BOOL* QueueIsFull,
                          const clocktype currentTime,
                          TosType* tos,
                          const double serviceTag = 0.0);
      virtual BOOL retrieve(Message** msg,
                            const int index,
                            const QueueOperation operation,
                            const clocktype currentTime,
                            double* serviceTag = NULL);
      virtual void finalize(Node* node,
                            const char* layer,
                            const int interfaceIndex,
                            const int instanceId,
                            const char* invokingProtocol = "IP",
                            const char* splStatStr = NULL);
};
```

FIGURE 4-107. Deriving a New Queue from `Queue` Class

4.4.7.4.6 Implementing Interface Functions

The next step in adding a new queue type is to implement the interface functions for the derived queue class (see Section 4.4.7.4.5). The derived queue class inherits the functions of the base class, but any additional functions that are declared in the derived queue class and any functions that override the base class functions need to be implemented. For the example queue, MYQUEUE, add these functions in the file queue_myqueue.cpp.

As an example, Figure 4-108 shows the implementation of the `RedQueue` function insert. The `RedQueue` function insert performs tasks specific to the RED queue model, and then calls the insert function of the base class `Queue`.

```
void RedQueue::insert(
    Message* msg,
    const void* infoField,
    BOOL* QueueIsFull,
    const clocktype currentTime,
    TosType* tos,
    const double serviceTag
    )
{
    ...
    if (!(MESSAGE_ReturnPacketSize(msg) <= (queueSizeInBytes - bytesUsed)))
    {
        // No space for this item in the queue
        *QueueIsFull = TRUE;
        // Update Generic Drop Stats
        numPacketsDropped++;
        numBytesDropped += MESSAGE_ReturnPacketSize(msg);
        return;
    }

    // Update Average Queue Size.
    UpdateAverageQueueSize(
        isEmpty(),
        packetsInQueue(),
        redParams->queueWeight,
        redParams->typicalSmallPacketTransmissionTime,
        startIdleTime,
        &averageQueueSize,
        currentTime);

    if (RedDropThePacket())
    {
         ...
        // A router has decided from its active queue management
        // mechanism, to drop a packet.
        *QueueIsFull = TRUE;

        // Update Generic Drop Stats
        numPacketsDropped++;
        numBytesDropped += MESSAGE_ReturnPacketSize(msg);
        return;
    }
    // Inserting a packet in the queue
     Queue::insert(msg,
                   infoField,
                   QueueIsFull,
                   currentTime,
                   serviceTag);
}
```

FIGURE 4-108. Implementation of an Interface Function

4.4.7.4.7 Modifying the Queue Setup Function

Function QUEUE_Setup, implemented in if_queue.cpp, is used by protocols to set up a queue. To add the example queue, MYQUEUE, modify QUEUE_Setup as shown in Figure 4-109.

```
void QUEUE_Setup(
   Node* node,
    Queue** queue,
    const char queueTypeString[],
    const int queueSize,
    const int interfaceIndex,
    const int queueNumber,
    const int infoFieldSize,
    const BOOL enableQueueStat,
    const BOOL showQueueInGui,
    const clocktype currentTime,
    const void* configInfo)
{
    if (!strcmp(queueTypeString, "FIFO"))
    {
       *queue = new Queue;
        (*queue)->SetupQueue(node,
                             queueTypeString,
                             queueSize,
                             interfaceIndex,
                             queueNumber,
                             infoFieldSize,
                             enableQueueStat,
                             showQueueInGui);
    }
    else if (!strcmp(queueTypeString, "RED"))
    {
        *queue = new RedQueue;
        ...
    }
    ...
    else if (!strcmp(queueTypeString, "MYQUEUE"))
    {
        *queue = new Myqueue;
        (*queue)->SetupQueue(node,
                             queueTypeString,
                             queueSize,
                             interfaceIndex,
                             queueNumber,
                             infoFieldSize,
                             enableQueueStat,
                             showQueueInGui,
                             ...);
    }
    ...
}
```

FIGURE 4-109. Modifying Function QUEUE_Setup

4.4.7.4.8 Including and Compiling Files

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.12).

4.4.7.4.9 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.4.8 EXata Schedulers

This section describes the scheduling schemes implemented in EXata. Section 4.4.8.1 and Section 4.4.8.2 describe the implementation of schedulers in EXata. Section 4.4.8.3 describes how EXata protocol models use the existing scheduler models. Section 4.4.8.4 describes the procedure to add a new scheduler model to EXata.

4.4.8.1 Data Structures and Classes

EXata implements several scheduling schemes (see Table 4-11). Schedulers are implemented using C++ classes. This section gives details of the base class used for deriving specific classes that implement different scheduling schemes. This section also describes some of the data structures used in the implementation of schedulers. These data structures and classes are defined in EXATA_HOME/include/if_scheduler.h. (Note that only a partial description of the data structures is provided here. Refer to the file if_scheduler.h for a complete description.)

1. `QueueData`**:** This structure contains information for one queue.

```
typedef struct queue_data_str
{
    Queue* queue;
    int priority;
    float weight;
    float rawWeight;
    char* infoField;
} QueueData;
```

   - `queue`: A pointer to an object of the `Queue` class implementing the queue.
   - `priority`: Priority of the queue.
   - `weight`: Weight of the queue.
   - `rawWeight`: Raw weight, i.e., without normalization.
   - `infoField`: User-specified data.

2. `Scheduler`: This is the base class that is used to derive specific scheduler classes. Figure 4-110, Figure 4-111, and Figure 4-112 show the declaration of the `Scheduler` class.

```
class Scheduler
{
    protected:
      QueueData*    queueData;
      int           numQueues;
      int           maxQueues;
      int           infoFieldSize;
      int           packetsLostToOverflow;

      // CurrentPeriod statistics
      clocktype               currentStateStartTime;
      clocktype               utilizedTime;
      BOOL                    stateIsIdle;
      int                     bytesDequeuedInPeriod;
      int                     packetsDequeuedInPeriod;
      clocktype               currentPeriodStartTime;
      clocktype               queueDelaysDuringPeriod;

      // Scheduler statistic collection
      BOOL                    schedulerStatEnabled;
      void*         schedGraphStatPtr;
      // Utility function for packet retrieval from specified priority queue
      QueueData*    SelectSpecificPriorityQueue(int priority);

    public:
       ...
}
```

FIGURE 4-110. Declaration of Class `Scheduler`: Protected Members

```
class Scheduler
{
    protected:
    ...

    public:
      void setRawWeight(const int priority, double rawWeight);
      void normalizeWeight();

      virtual int numQueue();
      virtual int GetQueuePriority(int queueIndex);

      virtual void insert(Message* msg,
                          BOOL* QueueIsFull,
                          const int priority,
                          const void* infoField,
                          const clocktype currentTime) = 0;

      virtual void insert(Message* msg,
                          BOOL* QueueIsFull,
                          const int priority,
                          const void* infoField,
                          const clocktype currentTime,
                          TosType* tos) = 0;

      virtual BOOL retrieve(const int priority,
                            const int index,
                            Message** msg,
                            int* msgPriority,
                            const QueueOperation operation,
                            const clocktype currentTime) = 0;

      virtual void setQueueBehavior(const int priority,
                                    QueueBehavior suspend = RESUME);

      virtual QueueBehavior getQueueBehavior(const int priority);

      virtual BOOL isEmpty(const int priority);

      virtual int bytesInQueue(const int priority);

      virtual int numberInQueue(const int priority);

      virtual int addQueue(Queue* queue,
                           const int priority = ALL_PRIORITIES,
                           const double weight = 1.0) = 0;

      virtual void removeQueue(const int priority) = 0;
      ...
```

FIGURE 4-111. Declaration of Class `Scheduler`: Public Members (Part 1)

```
class Scheduler
{
    protected:
      ...

    public:
      ...
      virtual void removeQueue(const int priority) = 0;

      virtual void swapQueue(Queue* queue, const int priority) = 0;

      virtual void qosInformationUpdate(int queueIndex,
                                int* qDelayVal,
                                int* totalTransmissionVal,
                                const clocktype currentTime,
                                BOOL isResetTotalTransmissionVal = FALSE);
      // Scheduler current period statistic collection
      //virtual int bytesDequeuedInPeriod(const int priority);
      //virtual clocktype utilizationInPeriod(const int priority);
      //virtual clocktype averageTimeInQueueDuringPeriod(const int priority);
      //virtual int resetPeriod(const clocktype currentTime);
      //virtual clocktype periodStartTime();

      // Scheduler statistic collection for graph
      virtual void collectGraphData(int priority,
                                    int packetSize,
                                    const clocktype currentTime);

      virtual void invokeQueueFinalize(Node* node,
                           const char* layer,
                           const int interfaceIndex,
                           const int instanceId,
                           const char* invokingProtocol = "IP",
                           const char* splStatStr = NULL);

      virtual void invokeQueueFinalize(Node* node,
                           const char* layer,
                           const int interfaceIndex,
                           const int instanceId,
                           const int fcsQosQueue,
                           const char* invokingProtocol = "IP",
                           const char* splStatStr = NULL);

      virtual void finalize(Node* node,
                           const char* layer,
                           const int interfaceIndex,
                           const char* invokingProtocol = "IP",
                           const char* splStatStr = NULL) = 0;

      // Virtual Destructor for Scheduler Class
      virtual ~Scheduler(){};
};
```

FIGURE 4-112. Declaration of Class `Scheduler`: Public Members (Part 2)

Some of the data members of the Scheduler class are described below. The utility functions of the Scheduler class are described in Section 4.4.8.2.

- queueData: List of queues controlled by the scheduler.
- numQueues: Number of queues instantiated by the scheduler.
- maxQueues: Current maximum size of queueData. This can be changed during execution.
- infoFieldSize: Size of infoField field of QueueData.
- packetsLostToOverflow: Total number of packets dropped by all queues controlled by the scheduler.
- currentStateStartTime: Start time of the current measurement period.
- schedulerStatEnabled: Indication whether statistics collection is enabled for the scheduler.
- schedGraphStatPtr: Pointer to the structure SchedGraphStat. This structure stores information for collecting and printing dynamic statistics.

4.4.8.2 Interface Functions

The protected interface function provided by the Scheduler class is listed below. This function is implemented in EXATA_HOME/libraries/developer/src/if_scheduler.cpp.

- SelectSpecificPriorityQueue: This function returns a pointer to the QueueData structure associated with the queue with the specified priority

In addition to the above function, the Scheduler class contains prototypes for the following virtual functions. Some of these functions for the base class are also implemented in the file if_scheduler.cpp. A class derived from the Scheduler class can provide alternate implementation of these functions and should provide implementations for the other virtual functions.

- numQueue: This function returns the number of queues associated with the scheduler.
- GetQueuePriority: This function returns the priority of the specified queue.
- insert: This function inserts a packet in the specified queue.
- retrieve: This function dequeues or takes a peek at a packet in the specified queue.
- isEmpty: This function determines if the queue with the specified priority is empty.
- bytesInQueue: This function returns the total number of bytes stored in a specific queue, or the total number of bytes stored in all queues associated with the scheduler.
- numberInQueue: This function returns the total number of messages in a specific queue, or the total number of bytes in all queues associated with the scheduler.
- addQueue: This function adds a queue to the scheduler.
- removeQueue: This function removes a queue from the scheduler.
- swapQueue: This function swaps a new queue and an existing queue with the specified priority. if no queue exists with the specified priority, this function adds a queue with that priority.
- qosInformatiopnUpdate: This function enables QOS monitoring for the specified queue.
- collectGraphData: This function enables performance data collection for the scheduler.
- invokeQueueFinalize: This function invokes the queue finalization function.
- finalize: This function outputs the final scheduler statistics.

In addition to the interface functions provided by the Scheduler class, function SCHEDULER_Setup is also available for implementing scheduling disciplines. Function SCHEDULE_Setup is used to create and initialize an object of the base Scheduler class or an object of a class derived from the base Scheduler class. The prototype of SCHEDULER_Setup is contained in if_scheduler.h and the implementation is

contained in if_scheuler.cpp. The prototype of SCHEDULER_Setup is shown below. See file if_scheduler.h for a description of the function parameters.

```
void SCHEDULER_Setup(
    Scheduler** scheduler,
    const char schedulerTypeString[],
    BOOL enableSchedulerStat = false,
    const char* graphDataStr = "NA");
```

4.4.8.3 Using the `Scheduler` Class

This section describes how protocols use the `Scheduler` class and interface functions to implement schedulers.

The `Scheduler` class can be used to implement schedulers at any layer. In this section, we illustrate the use of schedulers by taking code segments from the implementation of the IP protocol, which is implemented by files network_ip.h and network_ip.cpp in the folder EXATA_HOME/libraries/developer/src.

4.4.8.3.1 Creating and Initializing a Scheduler

To create a scheduler, declare a pointer variable that points to an object of the `Scheduler` class and call function SCHEDULER_Setup (see Section 4.4.8.2). The type of scheduler that is created is determined by the second parameter (`schedulerTypeString`) of SCHEDULER_Setup. Table 4-13 shows the different types of schedulers implemented in EXata. Other scheduler types can be derived from the `Scheduler` class (see Section 4.4.8.4).

TABLE 4-13. Implemented Scheduler Types

schedulerTypeString	Scheduler Type
"STRICT-PRIORITY"	Strict priority scheduler
"ROUND-ROBIN"	Round robin scheduler
"WEIGHTED-ROUND-ROBIN"	Weighted round robin scheduler
"WEIGHTED-FAIR"	Weighted fair queuing scheduler
"SELF-CLOCKED-FAIR"	Self-clocked fair queuing scheduler
"ATM"	Scheduler for ATM networks

As an example, Figure 4-113 shows the IP function NetworkIpInitCpuQueueConfiguration which initializes a strict priority scheduler by calling function SCHEDULER_Setup.

```
void
NetworkIpInitCpuQueueConfiguration(
    Node *node,
    const NodeInput *nodeInput)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    Scheduler *cpuSchedulerPtr = NULL;
    Queue* queue = NULL;
    queue = new Queue;
    BOOL enableQueueStat = FALSE;
    BOOL enableSchedulerStat = FALSE;
    char buf[MAX_STRING_LENGTH] = {0};
    BOOL wasFound = FALSE;
    int queueSize = DEFAULT_CPU_QUEUE_SIZE;
    if (ip->backplaneType == BACKPLANE_TYPE_CENTRAL)
    {
        queueSize = DEFAULT_CPU_QUEUE_SIZE * (node->numberInterfaces);
    }
    IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "INPUT-QUEUE-STATISTICS",
        &wasFound,
        buf);
    if (wasFound && (!strcmp(buf, "YES")))
    {
        enableQueueStat = TRUE;
    }
    IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "INPUT-SCHEDULER-STATISTICS",
        &wasFound,
        buf);
    if (wasFound && (!strcmp(buf, "YES")))
    {
        enableSchedulerStat = TRUE;
    }
    queue->SetupQueue(node, "FIFO", queueSize, 0, 0, 0, enableQueueStat);

    SCHEDULER_Setup(&cpuSchedulerPtr,
                    "STRICT-PRIORITY",
                    enableSchedulerStat);
    ip->cpuScheduler = cpuSchedulerPtr;
    // Scheduler add Queue Functionality
    cpuSchedulerPtr->addQueue(queue);
}
```

FIGURE 4-113. Creating and Initializing a Scheduler

4.4.8.3.2 Performing Scheduler Operations

Scheduler operations, such as adding and removing a queue and inserting and deleting a packet from a queue with a specified priority, are performed by calling the interface functions defined for the `Scheduler` class (see Section 4.4.8.2). The base class, `Scheduler`, implements some of the interface functions. The implementation of these functions can be found in if_scheduler.cpp. Implementation of the other interface functions are provided by the implementation of schedulers derived from the base class.

As an example, Figure 4-113 shows how the IP function NetworkIpInitCpuQueueConfiguration adds a queue by calling the `Scheduler` function addQueue. As another example, Figure 4-114 shows how the `Scheduler` function insert is used by the IP function NetworkIpQueueInsert to insert a packet in a queue controlled by the scheduler at the interface. NetworkIpInitCpuQueueConfiguration and NetworkIpQueueInsert are implemented in network_ip.cpp. Other `Scheduler` interface functions can be called in a similar way to perform scheduler operations.

```
void
NetworkIpQueueInsert(
    Node *node,
    Scheduler *scheduler,
    Message *msg,
    NodeAddress nextHopAddress,
    NodeAddress destinationAddress,
    int outgoingInterface,
    int networkType,
    BOOL *queueIsFull,
    int incomingInterface,
    BOOL isOutputQueue)
{
    int queueIndex = ALL_PRIORITIES;
    IpHeaderType *ipHeader = NULL;
    QueuedPacketInfo *infoPtr;
    BOOL isResolved = FALSE;

    ipHeader = (IpHeaderType*) MESSAGE_ReturnPacket(msg);

    // Tack on the nextHopAddress to the message using the insidious "info"
    // field.
    ...
    MESSAGE_InfoAlloc(node, msg, sizeof(QueuedPacketInfo));
    infoPtr = (QueuedPacketInfo *) MESSAGE_ReturnInfo(msg);
    ...
    // Call the Scheduler "insertFunction"
    queueIndex = GenericPacketClassifier(scheduler,
        (int) ReturnPriorityForPHB(node,
        IpHeaderGetTOS(ipHeader->ip_v_hl_tos_len)));

    (*scheduler).insert(msg,
                        queueIsFull,
                        queueIndex,
                        NULL, //const void* infoField,
                        getSimTime(node));
  }
```

FIGURE 4-114. Calling Interface Functions for Scheduler Operations

4.4.8.4 Adding a New Scheduler

This section describes the procedure to add a new scheduler to EXata. It describes how to derive a new scheduler class from the base class and how to implement interface functions.

We illustrate the process of adding a scheduler model by using as an example the implementation code for the strict priority scheduler. The header file for the strict priority scheduler implementation is sch_strictprio.h and the source file is sch_strictprio.cpp in the folder EXATA_HOME/libraries/developer/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a scheduler model. After understanding the discussed snippets, look at the complete code for the strict priority scheduler to understand how a scheduler model is derived from the base `Scheduler` class described in Section 4.4.8.1 and Section 4.4.8.2.

The following list summarizes the actions that need to be performed for adding a new scheduler model, MYSCHEDULER, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.4.8.4.1).
2. Modify the file if_scheduler.cpp to include the scheduler's header file (see Section 4.4.8.4.1).
3. Define data structures for the scheduler model (see Section 4.4.8.4.2).
4. Derive the new scheduler class based on the base `Scheduler` class (see Section 4.4.8.4.3).
5. Implement interface functions for the new scheduler model (see Section 4.4.8.4.4).
6. Modify function SCHEDULER_Setup to call MYSCHEDULER's constructor function (see Section 4.4.8.4.5)
7. Include the scheduler's header and source files in the EXata tree and compile (see Section 4.4.8.4.6).
8. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.4.8.4.7).

4.4.8.4.1 Creating Files

This step is similar to the one for Network Layer routing protocols (see Section 4.4.5.2). Create the header and source files for the scheduler model. Name these files in a way that clearly indicates the model that they implement. For scheduler models, prefix the file names with *sch_*.

Examples:

- sch_roundrobin.cpp, sch_roundrobin.h: These files in the folder EXATA_HOME/libraries/developer/src implement the round-robin scheduler.
- sch_strictprio.cpp, sch_strictprio.h: These files in the folder EXATA_HOME/libraries/developer/src implement the strict priority scheduler.

For the example scheduler, MYSCHEDULER, create files sch_myscheduler.h and sch_myscheduler.cpp in the folder EXATA_HOME/libraries/user_models/src (see Section 4.4.5.2).

It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems, even if the compilation process does not indicate any error.

While adding code to the files, it is important to organize the code well between the files. Generally, the header file for the scheduler model, sch_myscheduler.h, should contain the following:

- Constant definitions
- Data structure definitions
- Class definition for the new scheduler model derived from the base `Scheduler` class (see Section 4.4.8.1)
- Prototypes for any additional interface functions in the scheduler model's source file
- Statement to include the generic scheduler header file:

```
#include "if_scheduler.h"
```

The source file for the scheduler model, sch_myscheduler.cpp, should contain the following:

- Statement to include the scheduler model's header file:

```
#include "sch_myscheduler.h"
```

- Statements to include standard library functions and other header files needed by the queue model's source file. A typical queue model source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "api.h"          // EXATA_HOME/include/api.h
#include "sch_graph.h"
                    // EXATA_HOME/libraries/developer/src/sch_graph.h
```

- Interface functions for MYSCHEDULER

The file EXATA_HOME/libraries/developer/src/if_scheduler.cpp contains the generic function to setup a scheduler, SCHEDULER_Setup. SCHEDULER_Setup in turn calls constructor functions for specific scheduler types. To make the MYSCHEDULER constructor function available to function SCHEDULER_Setup, insert the following include statement in the file if_scheduler.cpp:

```
#include "sch_myscheduler.h"
```

4.4.8.4.2 Defining Data Structures

In EXata, scheduler types are defined as classes derived from the base `Scheduler` class. In addition to the variables that are part of `Scheduler` class, a scheduler type may require other variables. For example, the strict priority scheduler implementation uses the data structure `StrictPriorityStat`, defined in sch_strictprio.h, to store the statistic variables.

```
typedef struct
{
    unsigned int packetQueued;      // Total packet queued
    unsigned int packetDequeued;    // Total packet dequeued
    unsigned int packetDropped;     // Total packet dropped
}StrictPriorityStat;
```

For MYSCHEDULER, define an appropriate data structure for MYSCHEDULER-specific variables, if needed, in the file sch_myscheduler.h.

4.4.8.4.3 Deriving New Scheduler Class from Base `Scheduler` Class

In EXata all schedulers are implemented as classes derived from the base `Scheduler` class (see Section 4.4.8.1 and Section 4.4.8.2). This section describes how to implement a new scheduler as a class derived from the base `Scheduler` class by taking as an example the implementation of the strict priority scheduler.

Figure 4-115 shows the declaration of the class `StrictPriorityScheduler` that implements the strict priority scheduler. `StrictPriorityScheduler` is declared in sch_strictprio.h and is derived from the base class `Scheduler`. Class `StrictPriorityScheduler` declares a variable which is a pointer to the statistics data structure `StrictPriorityStat` (see Section 4.4.8.4.2) which stores the strict priority scheduler statistics. Class `StrictPriorityScheduler` also contains prototypes of interface functions that override the base `Scheduler` class functions. Implementation of these functions is discussed in Section 4.4.8.4.4.

For the example scheduler, MYSCHEDULER, declare a class `Myscheduler` derived from the base class `Scheduler` in the file sch_myscheduler.h.

```
class StrictPriorityScheduler : public Scheduler
{
    protected:
      StrictPriorityStat* stats;

    public:
      StrictPriorityScheduler(BOOL enableSchedulerStat,
                              const char graphDataStr[]);

      virtual ~StrictPriorityScheduler();

      virtual void insert(Message* msg,
                          BOOL* QueueIsFull,
                          const int queueIndex,
                          const void* infoField,
                          const clocktype currentTime);

      virtual void insert(Message* msg,
                          BOOL* QueueIsFull,
                          const int queueIndex,
                          const void* infoField,
                          const clocktype currentTime,
                          TosType* tos);

      virtual BOOL retrieve(const int priority,
                            const int index,
                            Message** msg,
                            int* msgPriority,
                            const QueueOperation operation,
                            const clocktype currentTime);

      virtual int addQueue(Queue* queue,
                           const int priority = ALL_PRIORITIES,
                           const double weight = 1.0);

      virtual void removeQueue(const int priority);

      virtual void swapQueue(Queue* queue, const int priority);

      virtual void finalize(Node* node,
                            const char* layer,
                            const int interfaceIndex,
                            const char* invokingProtocol = "IP",
                            const char* splStatStr = NULL);
};
```

FIGURE 4-115. Deriving a New Scheduler from `Scheduler` Class

4.4.8.4.4 Implementing Interface Functions

The next step in adding a new scheduler type is to implement the interface functions for the derived scheduler class (see Section 4.4.8.4.3). The derived scheduler class inherits the functions of the base class, but any functions that are declared in the derived queue class and any functions that override the base class functions need to be implemented. In addition, any virtual functions of the base class that are not implemented by the base class should be implemented in the derived class. For the example scheduler, MYSCHEDULER, add these functions in the file sch_myscheduler.cpp.

As an example, Figure 4-116 shows the implementation of the `StrictPriorityScheduler` function insert.

```
void StrictPriorityScheduler::insert(
    Message* msg,
    BOOL* QueueIsFull,
    const int priority,
    const void *infoField,
    const clocktype currentTime,
    TosType* tos
    )
{
    QueueData* qData = NULL;
    int queueIndex;
    int i;

    queueIndex = numQueues;
    for (i = 0; i < numQueues; i ++)
    {
        if (queueData[i].priority == priority)
        {
            queueIndex = i;
            break;
        }
    }
    ERROR_Assert((queueIndex >= 0) && (queueIndex < numQueues),
        "Queue does not exist!!!\n");

    // The priority queue in which incoming packet will be inserted
    qData = &queueData[queueIndex];
    ...
    // Insert the packet in the queue
    if(tos == NULL)
    {
        qData->queue->insert(msg, infoField, QueueIsFull, currentTime);
    }
    else
    {
        qData->queue->insert(msg, infoField, QueueIsFull, currentTime, tos);
    }

    if (!*QueueIsFull)
    {
        ...
        // Update packet enqueue status
        stats[queueIndex].packetQueued++;
    }
    else
    {
        ...
        // Update packet dequeue status
        stats[queueIndex].packetDropped++;
    }
}
```

FIGURE 4-116. Implementation of an Interface Function

4.4.8.4.5 Modifying the Scheduler Setup Function

Function SCHEDULER_Setup, implemented in if_scheduler.cpp, is used by protocols to set up a scheduler. To add the example scheduler, MYSCHEDULER modify SCHEDULER_Setup as shown in Figure 4-117.

```
void SCHEDULER_Setup(
    Scheduler** scheduler,
    const char schedulerTypeString[],
    BOOL enableSchedulerStat,
    const char* graphDataStr)
{
    if (!strcmp(schedulerTypeString, "STRICT-PRIORITY"))
    {
        *scheduler =
                new StrictPriorityScheduler(enableSchedulerStat, graphDataStr);
    }
    else if (!strcmp(schedulerTypeString, "ROUND-ROBIN"))
    {
        *scheduler =
               new RoundRobinScheduler(enableSchedulerStat, graphDataStr);
    }
    else if (!strcmp(schedulerTypeString, "MYSCHEDULER"))
    {
        *scheduler = new Myscheduler(enableSchedulerStat, graphDataStr);
    }
    ...
    if (*scheduler == NULL)
    {
        // Error:
        char errStr[MAX_STRING_LENGTH] = {0};
        sprintf(errStr, "Scheduler Error: Failed to assign memory for"
            " scheduler %s", schedulerTypeString);
        ERROR_ReportError(errStr);
    }
}
```

FIGURE 4-117. Modifying Function SCHEDULER_Setup

4.4.8.4.6 Including and Compiling Files

This step is similar to the one for adding a unicast routing protocol (see Section 4.4.5.12).

4.4.8.4.7 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.5 MAC Layer

The MAC Layer resides between the Network and Physical Layers in the EXata protocol stack, as shown in Figure 4-1. The MAC Layer provides error-free transfer of data across a link using the services of the Physical Layer. In EXata, some MAC protocols do not use the Physical Layer. Instead, the Physical Layer functionality is incorporated in the MAC protocol.

This section gives a detailed description of how to add a MAC Layer protocol to EXata.

4.5.1 MAC Layer Protocols in EXata

EXata provides a large number of MAC Layer protocols, both wired and wireless. If a node has multiple interfaces, it may run different MAC protocols at different interfaces. The MAC protocols running at different interfaces of a node can be a mix of wired and wireless protocols.

Besides different MAC protocols, EXata also simulates faults and switches at the MAC Layer. Table 4-14 lists the different MAC protocol models in EXata. Table 4-14 lists some of the protocols for switches in EXata. See the corresponding model library for a detailed description of each protocol and its parameters.

TABLE 4-14. MAC Protocols in EXata

MAC Protocol	Description	Model Library
ABSTRACT	Models the abstract MAC protocol for point-to-point links.	Developer
ABSTRACT-MAC	Models the abstract MAC protocol for cellular systems. (To select this protocol, parameter `MAC-PROTOCOL` should be set to `CELLULAR-MAC` and parameter `CELLULAR-MAC-PROTOCOL` should be set to `ABSTRACT-MAC`.)	Cellular
ALOHA	Models the ALOHA MAC protocol.	Wireless
CSMA	Models the Carrier Sense Multiple Access (CSMA) MAC protocol.	Wireless
GENERICMAC	Models an abstract wireless MAC protocol.	Wireless
GSM	Models the GSM MAC Layer.	Cellular
MAC-LTE	Models LTE Layer 2.	LTE
MAC802.3	Models the IEEE 802.3 MAC specification.	Developer
MAC802.15.4	Models the IEEE 802.15.4 MAC (ZigBee MAC) specification.	Sensor Networks
MAC802.16	Models the IEEE 802.16 MAC (WiMAX MAC) specification.	Advanced Wireless
MACA	Models the Multiple Access with Collision Avoidance (MACA) MAC protocol.	Wireless
MACDOT11	Models the IEEE 802.11 MAC specification. In addition, models the IEEE 802.11 MAC specification if 802.11 MAC is configured in infrastructure mode and the parameter `MAC-DOT11s-MESH-POINT` is set to `YES` for some node or interface in the subnet.	Wireless

TABLE 4-14. MAC Protocols in EXata (Continued)

MAC Protocol	Description	Model Library
MACDOT11e	Models the IEEE 802.11e MAC specification. This is a QoS enhancement to the IEEE 802.11 MAC. In addition, models the IEEE 802.11n MAC specification if `PHY-MODEL` is set to `PHY-802.11n`.	Wireless
SATCOM	Models an abstract MAC protocol for satellites.	Developer
SWITCHED-ETHERNET	Models an abstract switch connecting a subnet. This model does not have detailed models of switch ports, etc., and is limited to one subnet.	Multimedia and Enterprise
TDMA	Models the Time Division Multiple Access (TDMA) MAC protocol.	Wireless
UMTS-LAYER2	Models the abstract MAC protocol for UMTS systems. (To select this protocol, parameter `MAC-PROTOCOL` should be set to `CELLULAR-MAC` and parameter `CELLULAR-MAC-PROTOCOL` should be set to `UMTS-LAYER2`.)	UMTS

TABLE 4-15. Protocols for Switches in EXata

Protocol	Description	Network Type
GARP	Generic Attribute Registration Protocol	Multimedia and Enterprise
GVRP	GARP VLAN Registration Protocol	Multimedia and Enterprise
STP	Spanning Tree Protocol	Multimedia and Enterprise
VLAN	Virtual Local Area Network	Multimedia and Enterprise

4.5.2 MAC Layer Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to MAC Layer protocols. These files contain detailed comments on functions and other code components.

The MAC Layer API is composed of several macros, functions, and structures. These are defined in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/mac.h

  This file contains definitions common to MAC Layer protocols, the MAC data structure in the node structure, and prototypes of functions defined in EXATA_HOME/main/mac.cpp.
- EXATA_HOME/libraries/developer/src/network_ip.h

  This file contains definitions of some general API functions for interface address information operations.

- EXATA_HOME/include/phy.h
  This file contains definitions of API functions needed to communicate with the Physical Layer.

Additionally, the following header files are also relevant to the MAC Layer:

- EXATA_HOME/include/fileio.h
  This file contains prototypes of functions to read input files and create output files.
- EXATA_HOME/include/mapping.h
  This file contains prototypes of functions to map between node ids and IP addresses.

The following are the folders and source files associated with the MAC Layer:

- EXATA_HOME/libraries/developer/src, EXATA_HOME/libraries/developer/src
  These folders contain the source and header files for most of the MAC Layer protocols implemented in EXata. The file names are based on the name of the protocol that they implement, e.g., to see the implementation for IEEE 802.3, look at files mac_802_3.cpp and mac_802_3.h in the folder EXATA_HOME/libraries/developer/src. Other library folders may also contain code for other MAC protocols.
- EXATA_HOME/main/mac.cpp
  This file contains MAC Layer functions, including the initialization, message processing, and finalization functions.

4.5.3 MAC Layer Data Structures

The MAC Layer data structures are defined in EXATA_HOME/include/mac.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file mac.h for a complete description.)

1. `MAC_PROTOCOL`: This is an enumeration type that lists all the MAC Layer protocols.

```
typedef enum
{
    MAC_PROTOCOL_MPLS = 1,
    MAC_PROTOCOL_CSMA,
    MAC_PROTOCOL_FCSC_CSMA,
    MAC_PROTOCOL_MACA,
    MAC_PROTOCOL_FAMA,
    ...
    MAC_PROTOCOL_ABSTRACT,
    MAC_PROTOCOL_CELLULAR,
    MAC_PROTOCOL_ANE,
    MAC_PROTOCOL_WORMHOLE,
    MAC_PROTOCOL_ANODR,
    MAC_PROTOCOL_802_15_4,
    MAC_PROTOCOL_NONE // this must be the last one
} MAC_PROTOCOL;
```

2. `MacData`: This is the main data structure used by the MAC Layer and stores information about the MAC protocol running at a specific interface. Some important fields of this structure are explained below.

```
struct MacData
{
    MAC_PROTOCOL macProtocol;
    D_UInt32     macProtocolDynamic;
    int          interfaceIndex;
    BOOL         macStats;
    BOOL         promiscuousMode;
    ...
    Int64        bandwidth;   // In bytes.
    clocktype    propDelay;
    BOOL         interfaceIsEnabled;
    ...
    int          phyNumber;
    void         *macVar;
    void         *mplsVar;
    MacHasFrameToSendFn sendFrameFn;
    MacReceiveFrameFn receiveFrameFn;
    LinkedList  *interfaceStatusHandlerList;
    NodeAddress virtualMacAddress;
    MacVlan* vlan;
    void*       bgMainStruct; //ptr of background traffic main structure
    void*       randFault; //ptr of the random link fault data structure
    short       faultCount; //flag for link fault.
    MacHWAddress                macHWAddr;
    BOOL                         isLLCEnabled;
    BOOL                        interfaceCardFailed;
    ...
};
```

FIGURE 4-118. `MacData` Data Structure

- `macProtocol`: This is the MAC protocol running at the interface.
- `interfaceIndex`: This is the index of the interface.
- `macStats`: This Boolean variable indicates whether MAC statistics should be printed at end of simulation.
- `promiscuousMode`: This Boolean variable indicates whether the interface operates in promiscuous mode.
- `bandwidth`: This variable stores the bandwidth of the attached network.
- `propDelay`: This variable stores the propagation delay suffered by a packet.
- `phyNumber`: This is an index to the Physical Layer protocol running at the interface.
- `macVar`: This is a pointer to the data structure for the MAC protocol running at this interface.
- `sendFrameFn`: This is a pointer to the function used to send packets to the network.
- `receiveFrameFn`: This is a pointer to the function used to handle packets received from the network.
- `macHWAddr`: This is the MAC Layer address of the interface.

4.5.4 MAC Layer APIs and Inter-layer Communication

This section describes the APIs used by the Network Layer to communicate with the MAC Layer (see Section 4.5.4.1), the APIs used by the MAC Layer to communicate with the Network Layer (see Section 4.5.4.2), the APIs used by the MAC Layer protocols to communicate with the Physical Layer (see Section 4.5.4.3), and the APIs used by the Physical Layer to communicate with the MAC Layer (see Section 4.5.4.4). This section also lists some of the MAC Layer utility APIs (see Section 4.5.4.5).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.5.4.1 Network Layer to MAC Layer Communication

The API MAC_NetworkLayerHasPacketToSend is used by the Network Layer to inform the MAC Layer that a packet is available for transmission in the output queue.

When ARP is not enabled, the Network Layer calls function to IPv4AddresstoHWAddress to convert the IP address of the next hop of a packet to the MAC address used by the MAC protocol running at the interface.

The prototype for the functions MAC_NetworkLayerHasPacketToSend and IPv4AddresstoHWAddress are contained in the file mac.h. The functions are implemented in the file mac.cpp.

4.5.4.2 MAC Layer to Network Layer Communication

MAC Layer protocols use several APIs to communicate with the Network Layer. The prototypes for these functions are contained in the file mac.h. The file mac.cpp contains the implementation of these functions.

Some of the APIs used for communication from the MAC Layer to the Network Layer are listed below.

- MAC_OutputQueueIsEmpty: This functions checks if the output queue at an interface is empty.
- MAC_OutputQueueDequeuePacket: This function dequeues a packet from an output queue.
- MAC_OutputQueueTopPacket: This function is used to view the top packet of a queue without dequeuing it.
- MAC_OutputQueueDequeuePacketForAPriority: This function dequeues a specific priority packet from an output queue.
- MAC_HandOffSuccessfullyReceivedPacket: This function is used by the MAC Layer to pass a received packet to the upper layers.
- MAC_MacLayerAcknowledgement: This function notifies the Network Layer that a packet has been successfully delivered by the MAC protocol.
- MAC_NotificationOfPacketDrop: This function notifies the upper layer protocols when a packet is dropped at the MAC Layer.

4.5.4.3 MAC Layer to Physical Layer Communication

MAC Layer protocols use several APIs to communicate with the Physical Layer. The prototypes for these API functions are contained in the file EXATA_HOME/include/phy.h.

Some of the functions used for communication from the MAC Layer to the Physical Layer are listed below.

- PHY_StartTransmittingSignal: This function is used by the MAC Layer to send a packet to the Physical Layer.
- PHY_StartListeningToChannel: This function is used by the MAC Layer to direct the Physical Layer to start listening to the specified channel.
- PHY_StopListeningToChannel: This function is used by the MAC Layer to direct the Physical Layer to stop listening to the specified channel.

- PHY_SetTransmissionChannel: This function is used by the MAC Layer to set the channel for transmission.

4.5.4.4 Physical Layer to MAC Layer Communication

Physical Layer protocols use several APIs to communicate with the MAC Layer. The prototypes for these API functions are contained in the file mac.h. The file mac.cpp contains the implementation of these functions.

Some of the functions used for communication from the Physical Layer to the MAC Layer are listed below.

- MAC_ReceivePacketFromPhy: This function delivers a packet from the Physical Layer to the MAC Layer.
- MAC_ReceivePhyStatusChangeNotification: This function notifies the MAC Layer of a status change at the Physical Layer.

4.5.4.5 MAC Layer Utility APIs

Several APIs are available at the MAC Layer that perform tasks internal to the MAC Layer. Some of these functions can be used by other layers, as well. The prototypes for these API functions are contained in the file mac.h. The file mac.cpp contains the implementation of these functions.

Some of the MAC Layer utility APIs are listed below.

- MAC_IsMyHWAddress: This function checks whether a particular MAC address belongs to a specific interface of a node. The function returns TRUE if the address is a broadcast address or if the address is the unicast address of a specific interface of a node.
- MAC_IsMyAddress: This is an overloaded function which checks whether a particular address belongs to a specific node. One version of the function checks whether the address is the unicast address of any interface of the node. The other version checks whether the address is the unicast address of a specific interface of the node.
- MAC_IsBroadcastHWAddress: This function checks whether a particular address is a broadcast address.
- GetMacHWAddress: This function returns the MAC address of an interface of the node.
- MAC_IsWiredNetwork: This function checks if the attached network is a wired network.
- MAC_IsPointToPointNetwork: This function checks if the attached network is a point-to-point network.

4.5.5 Adding a Wired MAC Protocol

Although the working of each MAC protocol is different, there are certain functions that are performed by most MAC protocols. This section provides an overview of the flow of a MAC protocol for a wired network and provides an outline for developing and adding a MAC protocol, MYPROTOCOL, for a wired network to EXata. It describes how to develop code components common to most MAC protocols such as initializing, sending and receiving packets, and collecting statistics.

We illustrate the process of adding a MAC protocol for a wired network by using as an example the implementation code for the IEEE 802.3 protocol. The header file for the IEEE 802.3 implementation is mac_802_3.h and the source file is mac_802_3.cpp in the folder EXATA_HOME/libraries/developer/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a wired MAC protocol. After understanding the discussed snippets, look at the complete code for IEEE 802.3 to understand how a wired MAC protocol is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a wired MAC protocol to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.5.5.2).
2. Modify the file mac.cpp to include the protocol's header file (see Section 4.5.5.2).
3. Include the protocol in the list of MAC Layer protocols and trace protocols (see Section 4.5.5.3).
4. Define data structures for the protocol (see Section 4.5.5.4).
5. Decide on the format for the protocol-specific configuration parameters (see Section 4.5.5.5.1).
6. Read the protocol's configuration parameters and call the protocol's initialization function from the MAC Layer initialization function, MAC_Initialize (see Section 4.5.5.5.2).
7. Call the appropriate function to assign MAC addresses to interfaces (see Section 4.5.5.5.3). If MYPROTOCOL uses a new type of MAC address, then implement a function to assign MAC addresses of that type.
8. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Declare and initialize the state variables (see Section 4.5.5.5.4.1).
   b. Initialize send and receive function pointers (see Section 4.5.5.5.4.2).
   c. Initialize the neighbor list (see Section 4.5.5.5.4.3).
   d. Initialize times, if needed (see Section 4.5.5.5.4.4).
9. Implement functions to translate between IP and MAC addresses used by MYPROTOCOL (see Section 4.5.5.6).
10. Call the protocol's event dispatcher from the MAC Layer event dispatcher, MAC_ProcessEvent (see Section 4.5.5.7.1).
11. Declare any new event types used by the protocol in the header file EXATA_HOME/include/api.h (see Section 4.5.5.7.2).
12. Write the protocol event dispatcher (see Section 4.5.5.7.2).
13. Modify MAC Layer functions to integrate the new MAC protocol (see Section 4.5.5.8).
    a. Modify function MAC_NetworkLayerHasPacketToSend to deliver packets received from the Network Layer to MYPROTOCOL.
    b. Modify functions MAC_IsWiredNetwork, MAC_IsPointToPointNetwork, etc., to return the correct network type for the interface that MYPROTOCOL runs on.
14. Write a function to handle outgoing packets (see Section 4.5.5.9.1).
15. Write a function to process incoming packets (see Section 4.5.5.9.2).
16. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.5.5.10.1).
    b. Initialize the statistics variables in the protocol's initialization function (see Section 4.5.5.10.2).
    c. Update the statistics as appropriate (see Section 4.5.5.10.3).
    d. Write a function to print the statistics (see Section 4.5.5.10.4).
    e. Add dynamic statistics to the protocol, if desired (see Section 4.5.5.10.5) .
17. Call the protocol finalization function from the MAC Layer finalization function, MAC_Finalize (see Section 4.5.5.11.1).
18. Write the protocol finalization function (see Section 4.5.5.11.2). Call the function to print statistics from the protocol finalization function.

19. Include the protocol header and source files in the EXata tree and compile (see Section 4.5.5.12).

20. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.5.5.13).

4.5.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new protocol, the programmer name the different components of the new protocol in a similar manner. It will be helpful to examine the implementation of IEEE 802.3 in EXata for hints for naming and coding different components of the new protocol.

In this section, we describe the steps for developing a wired MAC protocol called "MYPROTOCOL". We will use the string "Myprotocol" in the names of the different components of this protocol, just as the string "Mac802_3" appears in the names of the components of the IEEE 802.3 implementation.

4.5.5.2 Creating Files

The first step towards adding a MAC protocol is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/developer/src. However, it is recommended that all user-developed models be made part of a library. In our example, we will place the MAC protocol in an a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn't already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the MAC protocol and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *mac_* to designate the files as MAC protocol files.

Examples*:*

- mac_802_3.h, mac_802_3.cpp: These files in the folder EXATA_HOME/libraries/developer/src implement the IEEE 802.3 MAC protocol.
- mac_tdma.h, mac_tdma.cpp: These files in the folder EXATA_HOME/libraries/wireless/src implement the TDMA MAC protocol.

In keeping with the naming guidelines of Section 4.5.5.1, the header file for the example protocol is called mac_myprotocol.h, and the source file is called mac_myprotocol.cpp.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, mac_myprotocol.h, should contain the following:

- Prototypes for interface functions in the source file, mac_myprotocol.cpp
- Constant definitions
- Data structure definitions and data types: `struct` and `enum` declarations

The source file, mac_myprotocol.cpp, should contain the following:

- Statement to include the protocol's header file:

```
#include "mac_myprotocol.h"
```

- Statements to include standard library functions and other header files needed by the protocol source file. A typical protocol source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "api.h"        // EXATA_HOME/include/api.h
#include "mac.h"        // EXATA_HOME/include/mac.h
#include "network_ip.h"
                  // EXATA_HOME/libraries/developer/src/network_ip.h
#include "partition.h" // EXATA_HOME/include/partition.h
```

- Initialization function for the protocol, MacMyprotocolInit
- Event dispatcher function for the protocol, MacMyprotocolLayer
- Finalization function for the protocol, MacMyprotocolFinalize
- Additional protocol implementation functions

The file mac.cpp contains the layer level initialization, event dispatcher, and finalization functions. These layer level functions in turn call the protocol's initialization, event dispatcher, and finalization functions. Therefore, to make these protocol functions available to the layer level functions, insert the following include statement in the file mac.cpp:

```
#include "mac_myprotocol.h"
```

4.5.5.3 Including MYPROTOCOL in List of MAC Layer Protocols

Each node in EXata hosts an operating protocol stack. For each layer in the stack, a list of protocols running at that layer is maintained. When a new MAC Layer protocol is added to EXata, it needs to be included in the list of MAC Layer protocols. To do this, add the protocol name to the enumeration MAC_PROTOCOL defined in mac.h (see Section 4.5.3).

For our example protocol, add the entry MAC_PROTOCOL_MYPROTOCOL to MAC_PROTOCOL, as shown in Figure 4-119.

```
typedef enum
{
    MAC_PROTOCOL_MPLS = 1,
    MAC_PROTOCOL_CSMA,
    MAC_PROTOCOL_FCSC_CSMA,
    MAC_PROTOCOL_MACA,
    MAC_PROTOCOL_FAMA,
    ...
    MAC_PROTOCOL_WORMHOLE,
    MAC_PROTOCOL_ANODR,
    MAC_PROTOCOL_802_15_4,
    MAC_PROTOCOL_MYPROTOCOL,
    MAC_PROTOCOL_NONE // this must be the last one
} MAC_PROTOCOL;
```

FIGURE 4-119. Adding MYPROTOCOL to List of MAC Layer Protocols

Note **Always add to the end of lists in header files.**

EXata provides for detailed traces of packets as they traverse the protocol stack at nodes in the network. A packet trace lists, among other information, the protocol that is handling the packet at the time of the trace. To facilitate tracing, EXata lists all protocols in an enumeration, `TraceProtocolType`, in the file EXATA_HOME/include/trace.h. For our example protocol, add an entry `TRACE_MYPROTOCOL` in `TraceProtocolType`, as shown in Figure 4-120.

```
typedef enum
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,                  // 1
    TRACE_UDP,                 // 2
    TRACE_IP,                  // 3
    TRACE_CBR,                 // 4
    ...
    TRACE_MYPROTCOL,
    // Must be last one!!!
    TRACE_ANY_PROTOCOL
}TraceProtocolType;
```

FIGURE 4-120. Adding MYPROTOCOL to List of Trace Protocols

Note **Always add to the end of lists in header files (just before the entry `TRACE_ANY_PROTOCOL`).**

4.5.5.4 Defining Data Structures

Each MAC Layer protocol has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Protocol parameters (see Section 4.5.5.5.2)
2. Protocol state (see Section 4.5.5.5.4.1)
3. Statistics variables (see Section 4.5.5.10.1)

Define an appropriate data structure, `MacDataMyprotocol`, for MYPROTCOL in the protocol header file, mac_myprotocol.h. As an example, the following data structure, defined in mac_802_3.h, is used by the IEEE 802.3 protocol:

```
typedef struct struct_mac_802_3_str
{
    MacData* myMacData;
    int bwInMbps;
    clocktype slotTime;
    clocktype interframeGap;
    clocktype jamTrDelay;
    Message* msgBuffer;
    MAC802_3StationState stationState;
    int backoffWindow;
    int collisionCounter;
    BOOL wasInBackoff;
    int seqNum;
    MAC802_3Statistics stats;
    LinkedList* neighborList;
    BOOL isFullDuplex;
    MAC802_3FullDuplexStatistics* fullDupStats;
    ...
} MacData802_3;
```

In the above declaration, `MAC802_3Statistics` is the statistics data structure and `MAC802_3StationState` is the enumeration of protocol states for the IEEE 802.3 protocol. See the declaration of `MacData802_3` in mac_802_3.h for a description of the other fields of the data structure.

4.5.5.5 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a wired MAC protocol.

4.5.5.5.1 Determining the Protocol Configuration Format

A MAC protocol may use protocol-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a MAC protocol's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

Generally, a wired MAC protocol requires the data rate and propagation delay to be specified. As an example, the following configuration specifies the subnet data rate and propagation delay as 10 Mbps and 1 micro-second, respectively.

```
SUBNET-DATA-RATE              10000000
SUBNET-PROPAGATION-DELAY      1US
```

Decide on the format for specifying the new protocol's configuration parameters. For our example protocol, specify the configuration parameters in the EXata configuration file using the following format (<Identifier> and <Index> can also be used to qualify the parameter declarations, as described above):

```
MAC-PROTOCOL          MYPROTOCOL
<param1>              <value1>
...
<paramN>              <valueN>
```

where

<param1>, ..., <paramN> : Names of parameters for MYPROTOCOL.

<value1>, ..., <valueN : Values of the protocol parameters.

Section 4.5.5.5.2 explains how to read user input specified in this format to initialize the protocol.

4.5.5.5.2 Reading Configuration Parameters and Calling the Protocol Initialization Function

EXata can configure a protocol to the parameters specified by the user in the EXata configuration file that sets up the experiment. This section explains how to read these user-specified configuration parameters for the MAC protocol and provide them to the protocol's initialization function.

The protocol stack of each node is initialized in a bottom up manner. For wired networks, the MAC Layer is the bottom-most layer and is initialized first. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the MAC Layer initialization function MAC_Initialize. Function MAC_Initialize reads the configuration file for lines starting with the keywords SUBNET or LINK (see Figure 4-121). If the input line begins with the keyword SUBNET, MAC_Initialize calls the function ProcessSubnetLine. If the input line begins with the keyword LINK, MAC_Initialize calls the function ProcessLinkLine.

```
void
MAC_Initialize(Node *firstNode,
               const NodeInput *nodeInput)
{
    ...
    char* subnetAddressString = (char*) MEM_malloc(MAX_ADDRESS_STRING_LENGTH);
    ...
    for (i = 0; i < nodeInput->numLines; i++)
    {
        char* currentLine = nodeInput->values[i];
        char *nextSubnetString = currentLine;
             BOOL isLink = FALSE;
        if (strcmp(nodeInput->variableNames[i], "SUBNET") == 0)
        {
        }
        else if (strcmp(nodeInput->variableNames[i], "LINK") == 0)
        {
            isLink = TRUE;
        }
        else
        {
            continue;
        }
        ...
        if (isLink == FALSE)
        {
           ProcessSubnetLine(firstNode,
                             nodeInput,
                             ipv4subnetAddress,
                             numHostBits,
                             &IPv6subnetAddress,
                             IPv6subnetPrefixLen,
                             p,
                             &subnetData->subnetList[subnetIndex],
                             subnetIndex,
                             siteCounter);
        }
        else
        {
            ProcessLinkLine(firstNode,
                            nodeInput,
                            p,
                            siteCounter);
         }
    }
    ...
}
```

FIGURE 4-121. Processing Input File `SUBNET` and `LINK` Statements in MAC_Initialize

Function ProcessSubnetLine assigns an IPv4 or IPv6 address to the subnet interface for each node in the subnet and calls function AddNodeToSubnet (AddNodeToIpv6Network) for each node. Function AddNodeToSubnet and AddNodeToIpv6Network initialize the interface information for the subnet interface and calls the initialization function for the MAC protocol specified for the subnet in the configuration file. For example, if IEEE 802.3 is specified as the MAC protocol, AddNodeToSubnet (see Figure 4-122) and AddNodeToIpv6Network (see Figure 4-123) call function CreateMac802_3. Functions MAC_Initialize, ProcessSubnetLine, ProcessLinkLine, AddNodeToSubnet, AddNodeToIpv6Network, and CreateMac802_3 are implemented in the file mac.cpp.

For some MAC protocols, function AddNodeToSubnet also updates the Network Layer forwarding table to include the routing information for nodes in the subnet. For IEEE 802.3, AddNodeToSubnet calls function NetworkUpdateForwardingTable to add an entry for nodes in the subnet.

```
static void //inline//
AddNodeToSubnet(
    Node *node,
    const NodeInput *nodeInput,
    NodeAddress subnetAddress,    //base address
    NodeAddress interfaceAddress, //my IP# on this interface
    int numHostBits,
    char *macProtocolName,
    PartitionSubnetMemberData* subnetList,
    int         nodesInSubnet,
    int         subnetListIndex,
    BOOL        isNewInterface,
    short       subnetIndex)
{
    int interfaceIndex;
    ...
    Address address;

    MacAddNewInterface(node, interfaceAddress, numHostBits, &interfaceIndex,
                       nodeInput, macProtocolName, isNewInterface);
    ...
    if(strcmp(macProtocolName, "MAC802.3") == 0 || ...)
    {
        /* Automatically add directly connected route to forwarding table. */
        NetworkUpdateForwardingTable(
                              node, subnetAddress,
                              ConvertNumHostBitsToSubnetMask(numHostBits),
                              0, interfaceIndex, 1, ROUTING_PROTOCOL_DEFAULT);
    }
    ...
    /* Select the MAC protocol and initialize. */
    if (strcmp(macProtocolName, "MAC802.3") == 0)
    {
        if (nodesInSubnet < 2)
        {
            char errorStr[MAX_STRING_LENGTH];
            sprintf(errorStr, "There are %d nodes (fewer than two) in subnet "
                              "%d", nodesInSubnet, subnetAddress);
            ERROR_ReportError(errorStr);
        }
        CreateMac802_3(node, &address, nodeInput, interfaceIndex,
                       (SubnetMemberData *) subnetList, nodesInSubnet,
                       NETWORK_IPV4);
        return;
    }
    else if (strcmp(macProtocolName, "SATCOM") == 0)
    {
        ...
    }
    ...
}
```

FIGURE 4-122. Wired MAC Initialization: Adding a Node to an IPv4 Subnet

```
static void //inline//
AddNodeToIpv6Network(
    Node *node,
    const NodeInput *nodeInput,
    in6_addr *globalAddr,
    in6_addr *subnetAddr,
    unsigned int subnetPrefixLen,
    char* macProtocolName,
    short subnetIndex,
    SubnetMemberData* subnetList,
    int nodesInSubnet,
    int subnetListIndex,
    unsigned short siteCounter,
    BOOL isNewInterface)
{
    int interfaceIndex;
    ...
    Address address;
    ...
    MacAddNewInterface(node, globalAddr, subnetAddr, subnetPrefixLen,
                       &interfaceIndex, nodeInput, siteCounter,
                       macProtocolName, isNewInterface);
    ...
    SetIPv6AddressInfo(&address, *globalAddr);
    ...
       /* Select the MAC protocol and initialize. */

    if (strcmp(macProtocolName, "MAC802.3") == 0)
    {
        if (nodesInSubnet < 2)
        {
            char errorStr[MAX_STRING_LENGTH];
            /*sprintf(errorStr, "There are %d nodes (fewer than two) in subnet"
                  " TLA=%u, NLA=%u, SLA=%u", nodesInSubnet, tla, nla, sla);*/
            ERROR_ReportError(errorStr);
        }
        CreateMac802_3(node, &address, nodeInput,
            interfaceIndex, subnetList, nodesInSubnet, NETWORK_IPV6);
        return;
    }
    else if (strcmp(macProtocolName, "SATCOM") == 0)
    {
        ...
    }
    ...
}
```

FIGURE 4-123. Wired MAC Initialization: Adding a Node to an IPv6 Subnet

Function CreateMac802_3, shown in Figure 4-124, reads the parameters for the protocol by calling function Mac802_3GetSubnetParameters, stores the parameters in the protocol data structure for the interface, `macData[interfaceIndex]`, initializes the Network Layer queues for the interface by calling function NetworkIpCreateQueues, and calls the initialization function for the IEEE 802.3 protocol, Mac802_3Init. NetworkIpCreateQueues is implemented in EXATA_HOME/libraries/developer/src/network_ip.cpp and the IEEE 802.3 functions are implemented in mac_802_3.cpp.

```
static void
CreateMac802_3(
    Node* node,
    Address* address,
    const NodeInput* nodeInput,
    int interfaceIndex,
    SubnetMemberData* subnetList,
    int nodesInSubnet,
    NetworkType networkType,
    BOOL fromLink = FALSE)
{
    Int64 subnetBandwidth;
    clocktype subnetPropDelay;
    ...
    Mac802_3GetSubnetParameters(node,
                                interfaceIndex
                                nodeInput,
                                address,
                                &subnetBandwidth,
                                &subnetPropDelay,
                                fromLink);

    if (fromLink)
    {
        SetMacConfigParameter(node,
                              interfaceIndex,
                              nodeInput);
    }

    NetworkIpCreateQueues(node, nodeInput, interfaceIndex);

    node->macData[interfaceIndex]->macProtocol = MAC_PROTOCOL_802_3;
    node->macData[interfaceIndex]->bandwidth = subnetBandwidth;
    node->macData[interfaceIndex]->propDelay = subnetPropDelay;
    node->macData[interfaceIndex]->mplsVar = NULL;

    Mac802_3Init(node,
                 nodeInput,
                 interfaceIndex,
                 subnetList,
                 nodesInSubnet);
}
```

FIGURE 4-124. MAC Protocol Initialization: Function CreateMac802_3

4.5.5.5.3 Initializing MAC Address

As part of the initialization of the MAC protocol at an interface, the interface is assigned a MAC address. This section describes how this is done for IPv4 networks. For IPv6 networks, the steps are very similar.

If a node is added to a subnet by means of the `SUBNET` keyword in the configuration file, function MAC_Initialize calls function ProcessInpuFileSubnetLine, which in turn calls function AddNodeToSubnet. (see Section 4.5.5.5.2). Function AddNodeToSubnet calls function MacAddNewInterface, which initializes the interface information for the node's interface to the subnet. MacAddNewInterface calls function MacConfigureHWAddress to initialize the MAC address for the interface.

MacConfigureHWAddress configures the hardware address at an interface of the node. MacAddNewInterface reads the MAC address configuration file if this interface address is not defined in the scenario configuration (.config) file. If the MAC address configuration file contains a MAC address for the interface being initialized, then that MAC address is assigned to the interface. Otherwise, MacConfigureHWAddress calls one of the following functions to assign a MAC address to the interface:

- MacSetDefaultHWAddress: To assign an Ethernet-type (6-byte) MAC address as a combination of node identifier and interface index
- MAC_SetFourByteMacAddress: To assign a 4-byte MAC address
- MAC_SetTwoByteMacAddress: To assign a 2-byte MAC address

Modify MacConfigureHWAddress to call the function to configure the MAC address for MYPROTOCOL, as shown in Figure 4-125. The example in Figure 4-125 assumes that MYPROTOCOL uses a 4-byte MAC address which is configured by function MAC_SetFourByteMacAddress.

If MYPROTOCOL supports a different type of MAC address, then you may need to implement a new MAC address generation function and call it from MacConfigureHWAddress.

```
static
void MacConfigureHWAddress(
    Node* node,
    NodeId nodeId,
    int interfaceIndex,
    const NodeInput* nodeInput,
    MacHWAddress* macAddr,
    char *macProtocolName,
    NetworkType networkType,
    Address ipAddr)
{
    ...
    macAddrConfigured = readInterfaceMacAddress(node, nodeid, interfaceIndex,
                                                nodeInput, macAddr,
                                                readMacAddress, ipAddr);
     if(!macAddrConfigured)
     {
          IO_ReadCachedFile(node, nodeid, interfaceIndex, nodeInput,
                            "MAC-ADDRESS-CONFIG-FILE", &retVal, &macAddrInput);
            ...
    }
   if (!macAddrConfigured)
    {
        if(strcmp(macProtocolName, "MAC802.3")==0 ||
           ...
           strcmp(macProtocolName, "SWITCHED-ETHERNET") == 0)
        {
                MacSetDefaultHWAddress(nodeId, macAddr, interfaceIndex);
        }
        else if(strcmp(macProtocolName, "MAC-LINK-11") == 0 ||
                strcmp(macProtocolName, "MYPROTOCOL") == 0  ||
                ...
                strcmp(macProtocolName, "MAC-SRW") == 0)
       {
             MAC_SetFourByteMacAddress(node, nodeId, macAddr, interfaceIndex);
        }
        else if(strcmp(macProtocolName, "MAC802.15.4") == 0)
        {
              MAC_SetTwoByteMacAddress(node, macAddr, interfaceIndex);
        }
        else
        {
                ERROR_Assert(FALSE,
                        "Invalid Mac Protocol or does not "
                        "have any supported Mac address type");
        }
    }
    ...
}
```

FIGURE 4-125. MAC Address Configuration Function

4.5.5.5.4 Implementing the Protocol Initialization Function

The initialization of a MAC protocol takes place in the initialization function of the protocol that is indirectly called by the MAC Layer initialization function MAC_Initialize. The initialization function of a wired MAC protocol commonly performs the following tasks:

- Create an instance of the protocol data structure
- Calculate and store the protocol's operational parameters
- Initialize the state variables
- Initialize the send and receive function pointers
- Collect neighbor information and create the neighbor list
- Schedule timers, if required

Like all other functions belonging to the protocol, the prototype for the initialization function should be included in the protocol's header file, mac_myprotocol.h.

4.5.5.5.4.1 Creating an Instance and Initializing the State

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as the protocol state and parameters, neighbor information, etc. Each instance of the protocol maintains its own state variable.

To store the state, declare the structure to hold the protocol state in the header file, mac_myprotocol.h (see Section 4.5.5.4). As an example, see the declaration of the IEEE 802.3 data structure `MacData802_3` in mac_802_3.h.

Create an instance of the protocol state by allocating memory to the state structure. IEEE 802.3 performs this task in its initialization function Mac802_3Init by calling the function MEM_malloc to allocate memory for the IEEE 802.3 data structure `MacData802_3`, as shown in Figure 4-126. Mac802_3Init and the other IEEE 802.3 functions are implemented in mac_802_3.cpp. Data structure and constant definitions for IEEE 802.3 are contained in mac_802_3.h.

Mac802_3Init also sets up pointers between the newly created instance of the IEEE 802.3 data structure `MacData802_3` and the data structure that stores the MAC Layer information for the interface, `macData[interfaceIndex]`.

The state variables for the protocol are also initialized in the initialization function. For example, Mac802_3Init initializes the protocol status (mode), backoff window, message buffer, etc.

The initialization function of a wired MAC protocol also calculates and stores the values of parameters that it requires in its operation. For example, function Mac802_3Init calculates and stores the values of the slot time, inter-frame gap and jam transmission time used by IEEE 802.3 by calling functions Mac802_3GetSlotTime, Mac802_3GetInterframeDelay, and Mac802_3GetJamTrDelay, respectively.

```
void Mac802_3Init(
    Node* node,
    const NodeInput* nodeInput,
    int interfaceIndex,
    SubnetMemberData* nodeList,
    int numNodesInSubnet)
{
    MacData802_3* mac802_3;
    ...
    mac802_3 = (MacData802_3 *) MEM_malloc(sizeof(MacData802_3));
    memset(mac802_3, 0, sizeof(MacData802_3));
    mac802_3->myMacData = node->macData[interfaceIndex];
    mac802_3->myMacData->macVar = (void *) mac802_3;
    RANDOM_SetSeed(mac802_3->seed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_802_3,
                   interfaceIndex);
    // Get channel bandwidth
    mac802_3->bwInMbps = (int) (mac802_3->myMacData->bandwidth / 1000000);
    // Initialize slot time, interframe gap and Jam transmission time.
    mac802_3->slotTime = Mac802_3GetSlotTime(node, mac802_3);
    mac802_3->interframeGap = Mac802_3GetInterframeDelay(node, mac802_3);
    mac802_3->jamTrDelay = Mac802_3GetJamTrDelay(node, mac802_3);
    // Initially there is no packet in own buffer
    mac802_3->msgBuffer = NULL;
    // Initialize state, collision counter
    // Backoff Window & Backoff Flag for this interface
    mac802_3->stationState = IDLE_STATE;
    mac802_3->collisionCounter = 0;
    mac802_3->wasInBackoff = FALSE;
    mac802_3->backoffWindow = MAC802_3_MIN_BACKOFF_WINDOW;
    mac802_3->seqNum = 0;
    ...
    // Initialize neighbor list for this station
    ListInit(node, &mac802_3->neighborList);
    mac802_3->link= (LinkData*)MEM_malloc(sizeof(LinkData));
    memset(mac802_3->link, 0, sizeof(LinkData));
    Mac802_3GetNeighborInfo(
        node,
        mac802_3->neighborList,
        nodeList,
        numNodesInSubnet,
        mac802_3->link);
    mac802_3->link->myMacData = mac802_3->myMacData;
    ...
}
```

FIGURE 4-126. IEEE 802.3 Initialization Function

4.5.5.5.4.2 Initializing Send and Receive Function Pointers

Each MAC protocol implements a function to transmit packets to the network and a function to handle packets received from the network. The pointers to these send and receive functions are stored in the `sendFrameFn` and `receiveFrameFn` fields, respectively, of the data structure that stores the MAC Layer information for the interface. This initialization is also done in the protocol initialization function. For example, the send and receive functions for the IEEE 802.3 protocol are Mac802_3NetworkLayerHasPacketToSend and Mac802_3HandleReceivedFrame, respectively (see Figure 4-126).

For MYPROTOCOL, write the send and receive functions MacMyprotocolNetworkLayerHasPacketToSend and MacMyprotocolHandleReceivedFrame (see Section 4.5.5.9) and set the send receive function pointers `sendFrameFn` and `receiveFrameFn` of the MAC data structure for the interface to point to these functions.

4.5.5.5.4.3 Initializing Neighbor List

For a wired MAC protocol, a node maintains a list of nodes to which it is connected through the subnet interface. This neighbor list is initialized in the protocol's initialization function. For example, for the IEEE 802.3 protocol, function Mac802_3Init calls function Mac802_3GetNeighborInfo (see Figure 4-126) to set up the neighbor list for a node.

4.5.5.5.4.4 Initializing Timers

A MAC protocol may need to set timers at initialization. See Section 3.3.2.2 for details on setting timers.

4.5.5.6 Implementing Address Translation Functions

Since the Network Layer and MAC Layer use different addresses for the same interface, the protocol implementation should provide for a way to translate from one address to another. Section 4.5.5.6.1 describes the function to translate an IP address to a MAC address. Section 4.5.5.6.2 describes the function to translate a MAC address to an IP address.

4.5.5.6.1 IP to MAC Address Translation Function

When the Network Layer passes a packet to the MAC Layer, it also passes the MAC address of the next hop. If ARP is not enabled, the Network Layer calls function IPv4AddressToHWAddress (for IPv4 networks) to translate the IP address of the next hop to the MAC address used by the MAC protocol running at the interface.

This section describes the translation from IPv4 to MAC addresses.

Function IPv4AddressToHWAddress translates an IP address to a MAC address based on the MAC protocol used by the interface. For Example, for the IEEE 802.3 protocol, function IPv4AddressToHWAddress calls function IPv4AddressToDefaultHWAddress, which translates an IPv4 address to a MAC address (see Figure 4-127). Modify function IPv4AddressToHWAddress to call the translation function for MYPROTOCOL. The example in Figure 4-127 assumes that MYPROTOCOL uses a 4-byte MAC address and calls MAC_FourByteMacAddressToVariableHWAddress which translates an IPv4 address to a MAC address. Functions IPv4AddressToHWAddress, IPv4AddressToDefaultHWAddress, and MAC_FourByteMacAddressToVariableHWAddress are implemented in mac.cpp.

If MYPROTOCOL supports a different type of MAC address, then you may need to implement a new MAC address to IPv4 address translation function.

```
BOOL IPv4AddressToHWAddress(
    Node *node,
    int interfaceIndex,
    Message* msg,
    NodeAddress ipv4Address,
    MacHWAddress* macAddr)
{
    BOOL isResolved = FALSE;
    switch (node->macData[interfaceIndex]->macProtocol)
    {
        ...
        case MAC_PROTOCOL_802_3:
        case MAC_PROTOCOL_ALOHA:
        ...
        case MAC_PROTOCOL_LINK:
        {
            (*macAddr).byte = (unsigned char*) MEM_malloc(
                      sizeof(unsigned char)*MAC_ADDRESS_DEFAULT_LENGTH);
            IPv4AddressToDefaultHWAddress(node, interfaceIndex,
                                          ipv4Address, macAddr);
            isResolved = TRUE;
            break;
        }

        case MAC_PROTOCOL_DOT11:
        ..
        case MAC_PROTOCOL_MYPROTOCOL:
        ...
        case MAC_PROTOCOL_CES_WINTGBS:
         {
                MAC_FourByteMacAddressToVariableHWAddress(node,
                                                          interfaceIndex,
                                                          macAddr,
                                                          ipv4Address);
                isResolved = TRUE;
                break;
         }
         ...
        default:
            ...
    }
    return isResolved;
}
```

FIGURE 4-127. Function to Translate IPv4 Addresses to MAC Addresses

4.5.5.6.2 MAC to IP Address Translation Function

When ARP is not enabled, some protocols may need to translate the MAC address of an interface to the IP address. This section describes the translation from MAC to IPv4 addresses.

Function MacHWAddressToIpv4Address translates a MAC address to an IP address based on the MAC address type. For example, for an Ethernet type address, function MacHWAddressToIpv4Address calls function DefaultMac802AddressToIpv4Address to translate the MAC address to IPv4 address (see Figure 4-127). Functions MacHWAddressToIpv4Address and DefaultMac802AddressToIpv4Address are implemented in mac.cpp.

If MYPROTOCOL supports a different type of MAC address, then you may need to implement a new IPv4 address to MAC address translation function.

```
NodeAddress MacHWAddressToIpv4Address(
                Node *node,
                int interfaceIndex,
                MacHWAddress* macAddr)
{
    if (MAC_IsBroadcastHWAddress(macAddr))
    {
            return ANY_DEST;
    }
    else if (ArpIsEnable(node,interfaceIndex))
    {
        return ReverseArpTableLookUp(node, interfaceIndex, macAddr);
    }
    else
    {
            switch(macAddr->hwType)
            {
                case HW_TYPE_ETHER:
                {
                      Mac802Address mac802Addr;
                      ConvertVariableHWAddressTo802Address(node,
                                                           macAddr,
                                                           &mac802Addr);
                      return DefaultMac802AddressToIpv4Address(node,
                                                               &mac802Addr);
                      break;
                }
                case IPV4_LINKADDRESS:
                {
                    return MAC_VariableHWAddressToFourByteMacAddress (
                                                               node,macAddr);
                    break;
                }
                case HW_NODE_ID:
                {
                    ...
                }
                default:
                    ERROR_Assert(FALSE, "Unsupported hardware type");
                    break;
            }
    }
    return 0;
}
```

FIGURE 4-128. Function to Translate MAC Addresses to IPv4 Addresses

4.5.5.7 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a wired MAC protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the MAC

Layer, NODE_ProcessEvent calls the MAC Layer event dispatcher MAC_ProcessEvent, defined in mac.cpp.

Section 4.5.5.7.1 describes how to modify the MAC Layer event dispatcher function to call the MAC protocol's event dispatcher. Section 4.5.5.7.2 describes how to implement the MAC protocol's event dispatcher.

4.5.5.7.1 Modifying the MAC Layer Event Dispatcher

Function MAC_ProcessEvent implements the MAC Layer event dispatcher that informs the appropriate MAC protocol of received events. Messages contain the index of the interface for which the event has occurred. The API function MESSAGE_GetInstanceId returns the interface index. MAC_ProcessEvent implements a switch statement on the protocol that is running at the interface read from the message and calls the appropriate protocol-specific event dispatcher.

To enable the protocol MYPROTOCOL to receive events, add code to MAC_ProcessEvent to call the protocol's event dispatcher function when messages for the protocol are received. Figure 4-129 shows a code fragment from MAC_ProcessEvent with sample code for calling MYPROTOCOL's event dispatcher function MacMyprotocolLayer.

```
void
MAC_ProcessEvent(Node *node, Message *msg)
{
    int interfaceIndex = MESSAGE_GetInstanceId(msg);
    int protocol = MESSAGE_GetProtocol(msg);
    ...
   /* Select the MAC protocol model, and direct it to handle the message. */

    switch (node->macData[interfaceIndex]->macProtocol)
    {
        ...
        case MAC_PROTOCOL_DOT11:
        {
            MacDot11Layer(node, interfaceIndex, msg);
            break;
        }
        case MAC_PROTOCOL_CSMA:
        {
            MacCsmaLayer(node, interfaceIndex, msg);
            break;
        }
        ...
        case MAC_PROTOCOL_MYPROTOCOL:
        {
            MacMyprotocolLayer(node, interfaceIndex, msg);
            break;
        }
        ...
     }
}
```

FIGURE 4-129. MAC Layer Event Dispatcher

4.5.5.7.2 Implementing the Protocol Event Dispatcher

A protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received.

All event types used by EXata protocols are enumerated in the file EXATA_HOME/include/api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as shown in Figure 4-130.

```
// /**
// ENUM        :: MESSAGE/EVENT
// DESCRIPTION :: Event/message types exchanged in the simulation
// **/
enum
{
    /* Special message types used for internal design. */
    MSG_SPECIAL_Timer                          = 0,
    ...
    /* Message Types for Channel layer */
    MSG_PROP_SignalArrival                     = 100,
    MSG_PROP_SignalEnd                         = 101,
    ...
    /*
     * Any other message types which have to be added should be added before
     * MSG_DEFAULT. Otherwise the program will not work correctly.
     */
    MSG_MAC_MYPROTOCOL_NewEvent1,
    MSG_MAC_MYPROTOCOL_NewEvent2,
    MSG_DEFAULT                                = 10000
};
```

FIGURE 4-130. Declaring New Event Types

Note **Always add to the end of lists in header files (just before the entry `MSG_DEFAULT`).**

The event dispatcher function for a MAC protocol handles both packet and timer events. As an example, Figure 4-131 shows the IEEE 802.3 event dispatcher function Mac802_3Layer. Note that once a message has been processed, it is freed by calling the API MESSAGE_Free, unless it is used to forward a packet to the upper layers. The event dispatcher also includes a default case in the switch statement to handle events of an unknown type.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

In Figure 4-131, event `MSG_MAC_StartTransmission` is an example of a timer event that is scheduled by one node for another node. Such timer events are used in wired MAC Protocols and Physical Layer models to model signal propagation. In IEEE 802.3, when a node starts transmitting a frame, it schedules a timer event of type `MSG_MAC_StartTransmission` to occur at each of its neighbor nodes after a delay that is equal to the propagation delay between the transmitting node and the receiving node. Event `MSG_MAC_TransmissionFinished` indicates the end of transmission of a frame. If the node is in the proper state (`RECEIVING_STATE`), this event also indicates that a packet has arrived at the node. Event `MSG_MAC_JamSequence` indicates the end of a jam sequence transmitted by a neighboring node. Event `MSG_MAC_TimerExpired` is a self timer, i.e., a timer that is scheduled by a node for itself.

See files mac_802_3.h and mac_802_3.cpp for definitions of data structures and functions used for implementing IEEE 802.3.

```
void Mac802_3Layer(
    Node* node,
    int interfaceIndex,
    Message* msg)
{
    MacData802_3* mac802_3 = (MacData802_3 *)
                              node->macData[interfaceIndex]->macVar;
    ...
    switch (msg->eventType)
    {
        case MSG_MAC_StartTransmission:
        {
            // Indicates some node has started to send a frame
           ...
        }
        case MSG_MAC_TransmissionFinished:
        {
            // Indicates a frame has come up to this node.
            ...
        }
        case MSG_MAC_JamSequence:
        {
           ...
        }
        case MSG_MAC_TimerExpired:
        {
            MAC802_3TimerType timerType;
            MAC802_3SelfTimer* info = NULL;
            // Get info from message
            info = (MAC802_3SelfTimer *) MESSAGE_ReturnInfo(msg);
            timerType = info->timerType;
            switch (timerType)
            {
                case mac_802_3_TimerSendPacket:
                {
                   ...
                }
                case mac_802_3_ChannelIdle:
                {
                   ...
                }
                ...
            }
            ...
        }
        default:
        {
            ...
        }
    }
}
```

FIGURE 4-131. IEEE 802.3 Event Dispatcher

4.5.5.8 Modifying MAC Layer Functions

The MAC Layer function MAC_NetworkLayerHasPacketToSend is used by the IP protocol, when the output queue is empty, to indicate to the MAC Layer that IP has a packet to send. MAC_NetworkLayerHasPacketToSend calls the send function for the MAC protocol running at the interface to process the packet from the Network Layer. To add a new MAC protocol, MYPROTOCOL, modify MAC_NetworkLayerHasPacketToSend so that the packet handler function for MYPROTCOL is called when MYPROTOCOL is running at the interface. MAC_NetworkLayerHasPacketToSend is implemented in mac.cpp. Figure 4-132 shows the changes that need to be made MAC_NetworkLayerHasPacketToSend. For IEEE 802.3, the function to handle packets received from the Network Layer is Mac802_3NetworkLayerHasPacketToSend (see Section 4.5.5.9). A pointer to this function is stored in the `sendFrameFn` field of the MAC data structure for that interface during initialization (see Section 4.5.5.5.4.2). The changes shown in Figure 4-132 assume that the initialization for MYPROTOCOL is done in the same way as for IEEE 802.3, described in Section 4.5.5.5.4, i.e., a pointer to the send function for MYPROTOCOL, MacMyprotocolNetworkLayerHaspacketToSend, is stored in `sendFrameFn`.

```
void
MAC_NetworkLayerHasPacketToSend(Node *node, int interfaceIndex)
{
    /* Select the MAC protocol model, and direct it to send/buffer the
       packet. */
    ...
    switch (node->macData[interfaceIndex]->macProtocol)
    {
         ...
        case MAC_PROTOCOL_802_3:
        {
            (*node->macData[interfaceIndex]->sendFrameFn)
                (node, interfaceIndex);
            break;
        }
        case MAC_PROTOCOL_MYPROTOCOL:
        {
            (*node->macData[interfaceIndex]->sendFrameFn)
                (node, interfaceIndex);
            break;
        }
        ...
     }
}
```

FIGURE 4-132. Delivering Packets from Network Layer to MAC Protocols

The MAC Layer also implements several functions that are used by upper layer protocols to determine the type of interface. These functions are:

- MAC_IsWiredNetwork: This function returns TRUE if the interface is a wired network.
- MAC_IsPointToPointNetwork: This function returns TRUE if the interface is a point-to-point link.
- MAC_IsWiredBroadcastNetwork: This function returns TRUE if the interface is a wired broadcast network.
- MAC_IsWirelessNetwork: This function returns TRUE if the interface is a wireless network.
- MAC_IsOneHopBroadcastNetwork: This function returns TRUE if the interface is a one-hop broadcast network.

These functions are implemented in mac.cpp. To add MYPROTOCOL to EXata, modify these functions so that they correctly indicate the type of network that MYPROTCOL runs on. As an example, the modification to function MAC_IsWiredNetwork is shown in Figure 4-133.

```
BOOL
MAC_IsWiredNetwork(Node *node, int interfaceIndex)
{
    // Abstract satellite model is considered a wired network
    // for the time being.

    if (node->macData[interfaceIndex]->macProtocol == MAC_PROTOCOL_LINK ||
//      node->macData[interfaceIndex]->macProtocol == MAC_PROTOCOL_MPLS ||
        node->macData[interfaceIndex]->macProtocol == MAC_PROTOCOL_802_3 ||
        node->macData[interfaceIndex]->macProtocol ==
                                                MAC_PROTOCOL_MYPROTOCOL ||
        node->macData[interfaceIndex]->macProtocol ==
                                                MAC_PROTOCOL_SWITCHED_ETHERNET)
    {
        return TRUE;
    }

    return FALSE;
}
```

FIGURE 4-133. Determining MAC Protocol Type

4.5.5.9 Interfacing with Network Layer

In this section we describe the interface between the Network Layer and a wired MAC Protocol.

A wired MAC protocol interacts with the Network Layer in the following ways:

1. When IP has a packet to send and the output queue is empty, IP indicates to the MAC protocol that a packet is ready for transmission. If the MAC protocol is in the appropriate state, it dequeues the packet from the output queue, adds a MAC header, and transmits the packet. See Section 4.5.5.9.1.
2. When the state of the MAC protocol changes to one where it can transmit a packet, the MAC protocol checks the output queue. If the queue is non-empty, the MAC protocol dequeues a packet from the queue, adds a MAC header, and transmits the packet. See Section 4.5.5.9.1.
3. When the MAC protocol receives a packet from the network that is meant for the Network Layer, the MAC protocol delivers the packet to the Network Layer. See Section 4.5.5.9.2.
4. When the MAC protocol receives a packet from the network that is not addressed to the node, but the node is operating in promiscuous mode, the MAC protocol delivers the packet to the Network Layer. See Section 4.5.5.9.2.
5. Some MAC protocols pass an indication to the Network Layer when certain events occur at the MAC Layer. These events include: a packet being dropped at the MAC Layer and receiving a MAC Layer acknowledgement for a transmitted packet. See Section 4.5.5.9.3.

4.5.5.9.1 Processing Outgoing Packets

When IP has a packet to send to the MAC Layer and the output queue is empty, IP calls function MAC_NetworkLayerHasPacketToSend. MAC_NetworkLayerHasPacketToSend calls the appropriate function for the MAC protocol running at the interface to process the packet from the Network Layer (see.Section 4.5.5.8). If IEEE 802.3 is running at the interface, MAC_NetworkLayerHasPacketToSend

calls the IEEE 802.3 function Mac802_3NetworkLayerHasPacketToSend. Mac802_3NetworkLayerHasPacketToSend, shown in Figure 4-134 and implemented in mac_802_3.cpp, checks the status of the node.

If the node's status is IDLE_STATE and its message buffer is empty, Mac802_3NetworkLayerHasPacketToSend does the following:

- Mac802_3NetworkLayerHasPacketToSend calls function Mac802_3RetrievePacketFromQIntoOwnBuffer to retrieve a packet from the output queue and add a MAC header to it by calling function Mac802_3CreateFrame.
- Mac802_3NetworkLayerHasPacketToSend then calls function Mac802_3SenseChannel to sense the channel and transmit the packet if the node's status is IDLE_STATE.

```
static void Mac802_3NetworkLayerHasPacketToSend(
    Node* node,
    int interfaceIndex)
{
    MacData802_3* mac802_3 = (MacData802_3 *)
                             node->macData[interfaceIndex]->macVar;
    ...
    if (mac802_3->stationState == IDLE_STATE)
    {
        // Check if there is any frame into own buffer
        if (mac802_3->msgBuffer != NULL)
        {
            // Frame present.
            // So no need to retrieve another.
            return;
        }

        // Retrieve the packet from queue into own buffer
        Mac802_3RetrievePacketFromQIntoOwnBuffer(node, mac802_3);

        // Sense the channel to transmit frame
        Mac802_3SenseChannel(node, mac802_3);
    }
}
```

FIGURE 4-134. Processing Outgoing Packets

For MYPROTOCOL, write a function, MacMyprotocolNetworkLayerHasPacketToSend, that performs appropriate actions to process an outgoing packet.

4.5.5.9.2 Processing Incoming Packets

The arrival of an incoming packet at a node is indicated by an event that is scheduled by the transmitting node. In IEEE 802.3, when a node completes transmission of a packet, it schedules event MSG_MAC_TransmissionFinished at all other nodes in the subnet to occur after the propagation delay.

Figure 4-135 shows the code segment from the IEEE 802.3 event handler function Mac802_3Layer that handles incoming packets. If the node's status is RECEIVING_STATE, then the event MSG_MAC_TransmissionFinished indicates that a packet has arrived at the node. Function Mac802_3Layer calls the receive function for IEEE 802.3, Mac802_3HandleReceivedFrame, to process the received frame. Note that a pointer to this function is stored in the receiveFrameFn field of the MAC data structure for that interface during initialization (see Section 4.5.5.5.4.2).

```
void Mac802_3Layer(
    Node* node,
    int interfaceIndex,
    Message* msg)
{
    MacData802_3* mac802_3 = (MacData802_3 *)
                             node->macData[interfaceIndex]->macVar;
    ...
    switch (msg->eventType)
    {
        ...
        case MSG_MAC_TransmissionFinished:
        {
            // Indicates a frame has come up to this node.
            if (mac802_3->stationState == RECEIVING_STATE)
            {
                // Frame received successfully.
                // Check the frame. If it is intended for this station
                // forward it to upper layer.
                (*mac802_3->myMacData->receiveFrameFn)(node,
                                                       interfaceIndex,
                                                       msg);
                if (mac802_3->wasInBackoff)
                {
                    // Previously, station was in Backoff state.
                    // So change station state as Backoff.
                    mac802_3->stationState = BACKOFF_STATE;
                }
                else
                {
                    // Station was Idle previously.
                    // Make station Idle and try to send next frame
                    // if available.
                    mac802_3->stationState = IDLE_STATE;
                    Mac802_3TryToSendNextFrame(node, mac802_3);
                }
                // Msg, containing the frame, will be freed properly later.
            }
            else
            {
                // Received a corrupted frame due to collision.
                // Discard this runt frame.
                MESSAGE_Free(node, msg);
            }
            break;
        }   ...
        ...
     }
}
```

FIGURE 4-135. Processing Incoming Packets

Function Mac802_3HandleReceivedFrame, shown in Figure 4-136, performs the following tasks:

- Mac802_3HandleReceivedFrame checks if the packet is addressed to the node.
- If the packet is addressed to the node, Mac802_3HandleReceivedFrame calls function Mac802_3ConvertFrameIntoPacket to remove the MAC header from the packet and delivers the packet to the upper layers by calling function MAC_HandoffSuccessfullyReceivedPacket.
- If the packet is not addressed to the node but the node is operating in promiscuous mode, Mac802_3HandleReceivedFrame calls function Mac802_3ConvertFrameIntoPacket to remove the MAC header from the packet and then calls function MAC_SneakPeakAtMacPacket to enable the Network Layer to examine the received packet.
- If the packet is not addressed to the node and the node is not operating in promiscuous mode, Mac802_3HandleReceivedFrame ignores the received packet.

```
static void Mac802_3HandleReceivedFrame(
    Node* node,
    int interfaceIndex,
    Message* msg)
{
    MacHeaderVlanTag tagInfo;
    unsigned short lengthOfPacket = 0;
    MacHWAddress srcHWAddr;
    MacHWAddress destHWAddr;

    MacData802_3* mac802_3 = (MacData802_3 *)
                             node->macData[interfaceIndex]->macVar;
    ...
    // Get destination and source address from the frame.
    Mac802_3GetSrcAndDestAddrFromFrame(node, msg, &destHWAddr, &srcHWAddr);
    ...
    BOOL isMyAddr = FALSE;
    if (NetworkIpIsUnnumberedInterface(node, interfaceIndex))
    {
        isMyAddr = (MAC_IsBroadcastHWAddress(&destHWAddr) ||
                    MAC_IsMyAddress(node, &destHWAddr));
    }
    else
    {
       isMyAddr = MAC_IsMyHWAddress(node, interfaceIndex, &destHWAddr);
    }
    // Checking whether the message is intended for this station
   if (isMyAddr)
    {
        // Frame intended for me. Increase numFrameReceived statistic
        mac802_3->stats.numFrameReceived++;
        Mac802_3Trace(node, mac802_3, msg, "R");
        Mac802_3ConvertFrameIntoPacket(node, msg, &tagInfo);
        ...
        MAC_HandOffSuccessfullyReceivedPacket(
            node, interfaceIndex, msg, &srcHWAddr);
    }
    // If node is operating in promiscuous mode then let
    // Network layer sneak a peak at the packet
    else if (node->macData[interfaceIndex]->promiscuousMode)
    {
        // Frame intended for me. So increase the numFrameReceived statistic.
        mac802_3->stats.numFrameReceived++;
        Mac802_3Trace(node, mac802_3, msg, "R");
        Mac802_3ConvertFrameIntoPacket(node, msg, &tagInfo);
        ...
        MAC_SneakPeekAtMacPacket(
            node, interfaceIndex, msg, srcAddr, destHWAddr);
        MESSAGE_Free(node, msg);
    }
    else
    {
        // Message for unknown destination. So ignore it.
        MESSAGE_Free(node, msg);
    }
}
```

FIGURE 4-136. Handling Received Packets

For MYPROTOCOL, write a function, MacMyprotocolHandleReceivedFrame, that performs appropriate actions to process a received packet.

4.5.5.9.3 Sending Indications to Network Layer

A MAC protocol may provide an indication to the Network Layer when the following events occur:

1. MAC protocol drops a packet: A MAC protocol may retransmit a packet up to a maximum number of times. When the maximum number of retransmissions is reached, the MAC protocol drops the packet and may inform the Network Layer of the dropped packet by calling function MAC_NotificationOfPacketDrop. See the implementation of IEEE 802.11 MAC in EXATA_HOME/libraries/wireless/src/mac_dot11-sta.h to see how this function is used.
2. MAC protocol receives an acknowledgement for a transmitted packet: When a MAC protocol receives an acknowledgement for a successfully transmitted packet, it may notify the Network Layer of the received acknowledgement by calling function MAC_MacLayerAcknowledgement. See the implementation of IEEE 802.11 MAC in mac_dot11-sta.h to see how this function is used.

4.5.5.10 Collecting and Reporting Statistics

In this section, we describe how to collect and report statistics for a MAC protocol.

4.5.5.10.1 Declaring Statistics Variables

A MAC Layer protocol can be configured to record statistics specified by the programmer, such as:

- Number of packets transmitted
- Number of packets received
- Number of packets discarded due to collision

To enable statistics collection for the protocol, include the statistic collection variables in the structure used to hold the protocol state (see Section 4.5.5.4). The statistics related variables can also be defined in a structure and then that structure is included in the state variable. For example, the data structure for IEEE 802.3, `MacData802_3`, contains the IEEE 802.3 statistics variable, `MAC802_3Statistics`, shown below:

```
typedef struct struct_mac_802_3_stat
{
    Int64 numFrameTransmitted;      // No of frame send by this station
    Int64 numFrameReceived;        // No of frame received by this station
    Int32 numBackoffFaced;          // No of times backoff faced
    Int64 numFrameLossForCollision;
                                  // No of frame discarded due to collision
} MAC802_3Statistics;
```

`MacData802_3` and `MAC802_3Statistics` are defined in mac_802_3.h.

4.5.5.10.2 Initializing Statistics

Initialize statistics variables in the protocol's initialization function. For example, the IEEE 802.3 initialization function Mac802_3Init, shown in Figure 4-137, initializes all fields of the statistics variable `MAC802_3Statistics` to 0.

```
void Mac802_3Init(
    Node* node,
    const NodeInput* nodeInput,
    int interfaceIndex,
    SubnetMemberData* nodeList,
    int numNodesInSubnet)
{
    ...
    // Initialize Stat Variables
    mac802_3->stats.numFrameTransmitted = 0;
    mac802_3->stats.numFrameReceived = 0;
    mac802_3->stats.numBackoffFaced = 0;
    mac802_3->stats.numFrameLossForCollision = 0;
    ...
}
```

FIGURE 4-137. Initializing Statistics Variables for IEEE 802.3

4.5.5.10.3 Updating Statistics

After declaring and initializing the statistics variables, update their value during the protocol life cycle, as required. For example, IEEE 802.3 increments the value of `numFrameTransmitted` in function Mac802_3CompleteFrameTransmission, implemented in mac_802_3.cpp, every time IEEE 802.3 transmits a packet, as shown in Figure 4-138.

```
static void Mac802_3CompleteFrameTransmission(
    Node* node,
    MacData802_3* mac802_3)
{
    ...
    // Send the message in the LAN, ie, to each neighbor
    Mac802_3BroadcastMessage(node,
                             mac802_3->msgBuffer,
                             mac802_3,
                             MSG_MAC_TransmissionFinished);

    // Station has sent the frame successfully.
    // Reset collision counter & Empty self buffer.

    // Increase numFrameTransmitted statistics
    mac802_3->stats.numFrameTransmitted++;
    ...
}
```

FIGURE 4-138. Updating IEEE 802.3 Statistics

4.5.5.10.4 Printing Statistics

As a final step towards statistics collection, create a function to print statistics. Call this function from the finalization function of the protocol, which is discussed in Section 4.5.5.11.2.

Function Mac802_3PrintStats, shown in Figure 4-139, calls the C function sprintf to create a single string containing the statistic name and statistic value, and then calls function IO_PrintStat to print that string to a file. Function IO_PrintStat function, defined in EXATA_HOME/include/fileio.h, requires the following parameters:

- Node pointer: Pointer to the node reporting the statistics.
- Layer: String indicating the layer. Set this to `"MAC"` for the MAC Layer.
- Protocol: String indicating the protocol name.
- Interface address: Interface address. Set this to `ANY_DEST` for MAC Layer protocols.
- Instance identifier: Interface index.
- Buffer: String containing the statistics.

```
static void Mac802_3PrintStats(
    Node* node,
    MacData802_3* mac802_3,
    int interfaceIndex)
{
    char buf[MAX_STRING_LENGTH];
    char buf1[MAX_STRING_LENGTH];

    ctoa(mac802_3->stats.numFrameTransmitted, buf1);
    sprintf(buf, "Number of Frames Transmitted = %s", buf1);
    IO_PrintStat(
        node,
        "MAC",
        "802.3",
        ANY_DEST,
        interfaceIndex,
        buf);

    ctoa(mac802_3->stats.numFrameReceived, buf1);
    sprintf(buf, "Number of Frames Received = %s", buf1);
    IO_PrintStat(
        node,
        "MAC",
        "802.3",
        ANY_DEST,
        interfaceIndex,
        buf);
   ...
}
```

FIGURE 4-139. Function to Print Statistics

4.5.5.10.5 Adding Dynamic Statistics

Dynamic statistics are statistic variables whose values can be observed in the EXata GUI during the simulation. See Section 5.2.3 for adding dynamic statistics to a protocol. Refer to *EXata User's Guide* for details of viewing dynamic statistics during the simulation.

4.5.5.11 Finalization

The finalization function of the protocol is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

At the end of simulation, the finalization function for each protocol is called to print the protocol statistics. As discussed in Section 3.4.3, the finalization function is called hierarchically. The node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp, calls the finalization function for MAC Layer, MAC_Finalize, defined in mac.cpp. MAC_Finalize calls the finalization function of the MAC protocol running at each interface.

4.5.5.11.1 Modifying the MAC Layer Finalization Function

Call the finalization function of the MAC protocol from the MAC Layer finalization function, MAC_Finalize, defined in mac.cpp. Figure 4-140 shows the outline of code that needs to be added to MAC_Finalize. Function MacMyprotocolFinalize is the finalization function of the protocol MYPROTOCOL (see Section 4.5.5.11.2).

```
void
MAC_Finalize(Node *node)
{
    int interfaceIndex;

    for (interfaceIndex = 0;
         interfaceIndex < node->numberInterfaces;
         interfaceIndex++)
    {
        ...
        /* Select the MAC protocol model and finalize it. */
        if (node->macData[interfaceIndex])
        {
            switch (node->macData[interfaceIndex]->macProtocol)
            {
                ...
                case MAC_PROTOCOL_802_3:
                {
                    Mac802_3Finalize(node, interfaceIndex);
                    break;
                }
                case MAC_PROTOCOL_MYPROTOCOL:
                {
                    MacMyprotocolFinalize(node, interfaceIndex);
                    break;
                }
                ...
            }
        }
        ...
}
```

FIGURE 4-140. MAC Layer Finalization Function

4.5.5.11.2 Implementing the Protocol Finalization Function

Write the finalization function for the protocol MYPROTOCOL, MacMyprotocolFinalize. If statistics collection is enabled for the MAC Layer, call the function to print the protocol's statistics (see Section 4.5.5.10.4) from the finalization function, or add code directly to MacMyprotocolFinalize to print statistics.

Use the IEEE 802.3 finalization function, Mac802_3Finalize, shown in Figure 4-141, as a template. Mac802_3Finalize is implemented in mac_802_3.cpp.

```
void Mac802_3Finalize(
    Node* node,
    int interfaceIndex)
{
    MacData802_3* mac802_3 = (MacData802_3 *)
                             node->macData[interfaceIndex]->macVar;

    if (node->macData[interfaceIndex]->macStats == TRUE)
    {
        // The mac can be either in full or half duplex mode
        // If its a full duplex, some extra statistics may be added
        if (mac802_3->isFullDuplex)
        {
            Mac802_3FullDuplexFinalize(node, mac802_3, interfaceIndex);
            // The existing 802.3 parameters may also be needed.
            //  If not needed then we may return from here.
        }
        else
        {
            Mac802_3PrintStats(node, mac802_3, interfaceIndex);
        }
    }
```

FIGURE 4-141. Finalization Function for IEEE 802.3

As for all other functions, specify the prototype of the finalization function in the protocol's header file, mac_myprotocol.h.

4.5.5.12 Including and Compiling Files

The final step in integrating your MAC protocol into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the MAC protocol in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Developer library, then modify EXATA_HOME/libraries/developer/Makefile-common as shown in Figure 4-142. Recompile EXata after making the changes.

```
...
# common sources
#
DEVELOPER_SRCS = \
$(DEVELOPER_SRCDIR)/adaptation_aal5.cpp \
$(DEVELOPER_SRCDIR)/adaptation.cpp \
...
$(DEVELOPER_SRCDIR)/mac_arp.cpp \
$(DEVELOPER_SRCDIR)/mac_llc.cpp \
$(DEVELOPER_SRCDIR)/mac_background_traffic.cpp \
$(DEVELOPER_SRCDIR)/mac_link.cpp \
$(DEVELOPER_SRCDIR)/mac_myprotocol.cpp \
$(DEVELOPER_SRCDIR)/mac_satcom.cpp \
$(DEVELOPER_SRCDIR)/mobility_placement.cpp \
$(DEVELOPER_SRCDIR)/multicast_igmp.cpp \
...
```

FIGURE 4-142. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.5.5.13 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.5.6 Adding a Wireless MAC Protocol

This section provides an overview of the flow of a MAC protocol for a wireless network and provides an outline for developing and adding a MAC protocol, MYPROTOCOL, for a wireless network to EXata. It describes how to develop code components common to most MAC protocols such as initializing, sending and receiving packets, and collecting statistics.

We illustrate the process of adding a MAC protocol for a wireless network by using as an example the implementation code for the CSMA protocol. The header file for the CSMA implementation is mac_csma.h and the source file is mac_csma.cpp in the folder EXATA_HOME/libraries/wireless/src. We use code snippets from these two files throughout this section to illustrate different steps in writing a wireless MAC protocol. After understanding the discussed snippets, look at the complete code for the CSMA protocol to understand how a wireless MAC protocol is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a wireless MAC protocol, MYPROTOCOL, to EXata. For those steps that are similar to the steps for writing a wired MAC protocol, we refer the reader to the appropriate subsection of Section 4.5.5. The steps that are different for wired MAC protocols are described in detail in subsequent sections.

1. Create header and source files (see Section 4.5.5.2).
2. Modify the file mac.cpp to include the protocol's header file (see Section 4.5.5.2).
3. Include the protocol in the list of MAC Layer protocols and trace protocols (see Section 4.5.5.3).
4. Define data structures for the protocol (see Section 4.5.6.1).
5. Decide on the format for the protocol-specific configuration parameters (see Section 4.5.6.2.1).

6. Call the protocol's initialization function from the MAC Layer initialization function, MAC_Initialize (see Section 4.5.6.2.2).
7. Call the appropriate function to assign MAC addresses to interfaces (see Section 4.5.6.2.3). If MYPROTOCOL uses a new type of MAC address, then implement a function to assign MAC addresses of that type.
8. Write the initialization function for the protocol. The initialization function should include the following tasks:
   a. Declare and initialize the state variables (see Section 4.5.6.2.4.1).
   b. Read and store the configuration parameters (see Section 4.5.6.2.4.1).
   c. Initialize times, if needed (see Section 4.5.6.2.4.2).
9. Implement functions to translate between IP and MAC addresses used by MYPROTOCOL (see Section 4.5.6.3).
10. Call the protocol's event dispatcher from the MAC Layer event dispatcher, MAC_ProcessEvent (see Section 4.5.6.4.1).
11. Declare any new event types used by the protocol in the header file EXATA_HOME/include/api.h (see Section 4.5.6.4.2).
12. Write the protocol event dispatcher (see Section 4.5.6.4.2).
13. Modify MAC Layer functions to integrate the new MAC protocol (see Section 4.5.6.5).
   a. Modify function MAC_NetworkLayerHasPacketToSend to deliver packets received from the Network Layer to MYPROTOCOL.
   b. Modify functions MAC_IsWirelessNetwork, MAC_IsOneHopBroadcastNetwork, etc., to return the correct network type for the interface that MYPROTOCOL runs on.
   c. Modify function MAC_ReceivePacketFromPhy to deliver packets received from the Physical Layer to MYPROTOCOL.
   d. Modify function MAC_ReceivePhyStatusChangeNotification to deliver Physical Layer status change notifications to MYPROTOCOL.
14. Write a function to handle outgoing packets (see Section 4.5.6.6.1).
15. Write a function to process incoming packets (see Section 4.5.6.6.2).
16. Write a function to process Physical Layer status changes (see Section 4.5.6.6.3).
17. Include code in various functions to collect statistics.
   a. Declare statistics variables (see Section 4.5.5.10.1).
   b. Initialize the statistics variables in the protocol's initialization function (see Section 4.5.5.10.2).
   c. Update the statistics as appropriate (see Section 4.5.5.10.3).
   d. Write a function to print the statistics (see Section 4.5.5.10.4).
   e. Add dynamic statistics to the protocol, if desired (see Section 4.5.5.10.5) .
18. Call the protocol finalization function from the MAC Layer finalization function, MAC_Finalize (see Section 4.5.5.11.1).
19. Write the protocol finalization function (see Section 4.5.5.11.2). Call the function to print statistics from the protocol finalization function.
20. Include the protocol header and source files in the EXata tree and compile (see Section 4.5.5.12).
21. To make the protocol available in the EXata GUI, modify the GUI settings files (see Section 4.5.6.10).

4.5.6.1 Defining Data Structures

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.4).

Each MAC Layer protocol has its own data structures, which are defined in the protocol's header file. The data structures store information such as:

1. Protocol parameters (see Section 4.5.6.2.4)
2. Protocol state (see Section 4.5.6.2.4)
3. Statistics variables (see Section 4.5.5.10.1)

Define an appropriate data structure, `MacDataMyprotocol`, for MYPROTCOL in the protocol header file, mac_myprotocol.h. As an example, the following data structure, defined in mac_csma.h, is used by the CSMA protocol:

```
typedef struct struct_mac_csma_str
{
    MacData* myMacData;
    Int32 status;           /* status of layer CSMA_STATUS_* */
    Int32 BOmin;            /* minimum backoff */
    Int32 BOmax;            /* maximum backoff */
    Int32 BOtimes;          /* how many times has it backoff ? */
    Int32 pktsToSend;
    Int32 pktsLostOverflow;
    Int32 pktsSentUnicast;
    Int32 pktsSentBroadcast;
    Int32 pktsGotUnicast;
    Int32 pktsGotBroadcast;
    CsmaTimer timer;
    RandomSeed seed;        /* for setting backoff timer */
} MacDataCsma;
```

4.5.6.2 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a wireless MAC protocol.

4.5.6.2.1 Determining the Protocol Configuration Format

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.5.1).

4.5.6.2.2 Calling the Protocol Initialization Function

The protocol stack of each node is initialized in a bottom up manner. For wireless networks, the MAC Layer at an interface is initialized after the Physical Layer model for the interface is initialized. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the MAC Layer initialization function MAC_Initialize. Function MAC_Initialize reads the configuration file for lines starting with the keywords `SUBNET` or `LINK`. If the input line begins with the keyword `SUBNET`, MAC_Initialize calls function ProcessSubnetLine. If the input line begins with the keyword `LINK`, MAC_Initialize calls function ProcessLinkLine. Function ProcessSubnetLine assigns an IPv4 and/or IPv6 address to the subnet interface for each node in the subnet and calls function AddNodeToSubnet or AddNodeToIpv6Network for each node. Functions AddNodeToSubnet and AddNodeToIpv6Network initialize the interface information

for the subnet interface. Functions MAC_Initialize, ProcessSubnetLine, ProcessLinkLine, AddNodeToSubnet, and AddNodeToIpv6Network are implemented in the file mac.cpp.

For a wireless MAC protocol, function AddNodeToSubnet (AddNodeToIpv6Network) initializes the Physical Layer model specified for the interface. AddNodeToSubnet (AddNodeToIpv6Network) then determines the data rate for the index by calling function PHY_GetTxDataRate and initializes the Network Layer queues for the interface by calling function NetworkIpCreateQueues. AddNodeToSubnet (AddNodeToIpv6Network) then calls the initialization function for the MAC protocol running at the interface. For example, if CSMA is specified as the MAC protocol, AddNodeToSubnet (AddNodeToIpv6Network) calls the CSMA initialization function, function MacCsmaInit (see Figure 4-143). PHY_GetTxDataRate is implemented in EXATA_HOME/libraries/wireless/src/phy.cpp and NetworkIpCreateQueues is implemented in EXATA_HOME/libraries/developer/src/network_ip.cpp. MacCsmaInit and the other CSMA functions are implemented in mac_csma.cpp. Modify AddNodeToSubnet (AddNodeToIpv6Network) to call the MYPROTOCOL initialization function, MacMyprotocolInit, if MYPROTOCOL is specified as the MAC protocol for the interface, as shown in Figure 4-143 (Figure 4-144).

```
    static void //inline//
    AddNodeToSubnet(
        Node *node,
        const NodeInput *nodeInput,
        NodeAddress subnetAddress,    //base address
        NodeAddress interfaceAddress, //my IP# on this interface
        int numHostBits,
        char *macProtocolName,
        PartitionSubnetMemberData* subnetList,
        int         nodesInSubnet,
        int         subnetListIndex,
        BOOL        isNewInterface,
        short       subnetIndex)
    {
        int interfaceIndex;
        ...
        Address address;

        MacAddNewInterface(node, interfaceAddress, numHostBits, &interfaceIndex,
                           nodeInput, macProtocolName, isNewInterface);
        ...
        // bandwidth is set to the base data rate
        // (in case it's variable)
        //
        node->macData[interfaceIndex]->bandwidth =
            (PHY_GetTxDataRate(
                node,
                node->macData[interfaceIndex]->phyNumber) / 8);

        NetworkIpCreateQueues(node, nodeInput,
                              interfaceIndex);

       if (strcmp(macProtocolName, "CSMA") == 0) {
            node->macData[interfaceIndex]->macProtocol = MAC_PROTOCOL_CSMA;
            MacCsmaInit(node, interfaceIndex, nodeInput);
        }
            else if (strcmp(macProtocolName, "MYPROTOCOL") == 0) {
               node->macData[interfaceIndex]->macProtocol =
                                                    MAC_PROTOCOL_MYPROTOCOL;
                MacMyprotocolInit(node, interfaceIndex, nodeInput);
            }
        ...
}
```

FIGURE 4-143. Wireless MAC Initialization: Adding a Node to an IPv4 Subnet

```
    static void //inline//
    AddNodeToIpv6Network(
        Node *node,
        const NodeInput *nodeInput,
        in6_addr *globalAddr,
        in6_addr *subnetAddr,
        unsigned int subnetPrefixLen,
        char* macProtocolName,
        short subnetIndex,
        SubnetMemberData* subnetList,
        int nodesInSubnet,
        int subnetListIndex,
        unsigned short siteCounter,
        BOOL isNewInterface)
    {
        int interfaceIndex;
        ...
        Address address ;

        MacAddNewInterface(node, globalAddr, subnetAddr, subnetPrefixLen,
                           &interfaceIndex, nodeInput, siteCounter, macProtocolName,
                            isNewInterface);
        ...
        // bandwidth is set to the base data rate
        // (in case it's variable)
        //
        node->macData[interfaceIndex]->bandwidth =
            (PHY_GetTxDataRate(
                node,
                node->macData[interfaceIndex]->phyNumber) / 8);
        NetworkIpCreateQueues(node, nodeInput,
                              interfaceIndex);
        if (strcmp(macProtocolName, "CSMA") == 0) {
            node->macData[interfaceIndex]->macProtocol = MAC_PROTOCOL_CSMA;
            MacCsmaInit(node, interfaceIndex, nodeInput);
        }
        else if (strcmp(macProtocolName, "MYPROTOCOL") == 0) {
              node->macData[interfaceIndex]->macProtocol =
                                                    MAC_PROTOCOL_MYPROTOCOL;
              MacMyprotocolInit(node, interfaceIndex, nodeInput);
        }
        ...
}
```

FIGURE 4-144. Wireless MAC Initialization: Adding a Node to an IPv6 Subnet

4.5.6.2.3 Initializing MAC Address

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.5.3).

4.5.6.2.4 Implementing the Protocol Initialization Function

The initialization of a wireless MAC protocol takes place in the initialization function of the protocol that is indirectly called by the MAC Layer initialization function MAC_Initialize. The initialization function of a wireless MAC protocol commonly performs the following tasks:

- Create an instance of the protocol data structure
- Calculate and store the protocol's operational parameters
- Initialize the state variables
- Schedule timers, if required

Like all other functions belonging to the protocol, the prototype for the initialization function should be included in the protocol's header file, mac_myprotocol.h.

4.5.6.2.4.1 Creating an Instance and Reading Configuration Parameters

The initialization function initializes the protocol state. Each protocol has a structure that it uses to store state information. This may include information such as the protocol state and parameters, statistics variables, etc. Each instance of the protocol maintains its own state variable.

To store the state, declare the structure to hold the protocol state in the header file, mac_myprotocol.h (see Section 4.5.5.4). As an example, see the declaration of the CSMA data structure `MacDataCsma` in mac_csma.h.

Create an instance of the protocol state by allocating memory to the state structure. CSMA performs this task in its initialization function MacCsmaInit by calling the function MEM_malloc to allocate memory for the CSMA data structure `MacDataCsma`, as shown in Figure 4-145. MacCsmaInit and the other CSMA functions are implemented in mac_csma.cpp. Data structure and constant definitions for CSMA are contained in mac_csma.h.

MacCsmaInit also sets up pointers between the newly created instance of the CSMA data structure `MacDataCsma` and the data structure that stores the MAC Layer information for the interface, `macData[interfaceIndex]`.

The state variables for the protocol are also initialized in the initialization function. For example, MacCsmaInit initializes the protocol status (mode), backoff parameters, etc.

The initialization function of a wireless MAC protocol may also calculate and store the values of parameters that it requires in its operation. If the protocol has any user-specified configuration parameters, these are read in the protocol initialization function. CSMA does not have any user-specified configuration parameters. To understand how configuration parameters are read from an input file, refer to the IEEE 802.3 function Mac802_3GetSubnetParameters (see Section 4.5.5.5.2) or the IEEE 802.11 MAC initialization function MacDot11Init in EXATA_HOME/libraries/wireless/src//mac_dot11.cpp.

```
void MacCsmaInit(
   Node *node, int interfaceIndex, const NodeInput *nodeInput)
{
    MacDataCsma *csma = (MacDataCsma *) MEM_malloc(sizeof(MacDataCsma));
    assert(csma != NULL);

    memset(csma, 0, sizeof(MacDataCsma));
    csma->myMacData = node->macData[interfaceIndex];
    csma->myMacData->macVar = (void *)csma;
    csma->timer.flag = CSMA_TIMER_OFF | CSMA_TIMER_UNDEFINED;
    csma->timer.seq = 0;
    csma->status = CSMA_STATUS_PASSIVE;
    csma->BOmin = CSMA_BO_MIN;
    csma->BOmax = CSMA_BO_MAX;
    csma->BOtimes = 0;
    csma->pktsToSend = 0;
    csma->pktsLostOverflow = 0;
    csma->pktsSentUnicast = 0;
    csma->pktsSentBroadcast = 0;
    csma->pktsGotUnicast = 0;
    csma->pktsGotBroadcast = 0;
    RANDOM_SetSeed(csma->seed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_CSMA,
                   interfaceIndex);
   ...
}
```

FIGURE 4-145. CSMA Initialization Function

4.5.6.2.4.2 Initializing Timers

A MAC protocol may need to set timers at initialization. See Section 3.3.2.2 for details on setting timers.

4.5.6.3 Implementing Address Translation Functions

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.6).

4.5.6.4 Implementing the Event Dispatcher

In this section, we describe the steps for implementing the event dispatcher function for a wireless MAC protocol.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the MAC Layer, NODE_ProcessEvent calls the MAC Layer event dispatcher MAC_ProcessEvent, defined in mac.cpp.

Section 4.5.6.4.1 describes how to modify the MAC Layer event dispatcher function to call the MAC protocol's event dispatcher. Section 4.5.6.4.2 describes how to implement the MAC protocol's event dispatcher.

4.5.6.4.1 Modifying the MAC Layer Event Dispatcher

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.7.1).

4.5.6.4.2 Implementing the Protocol Event Dispatcher

A protocol's event dispatcher should include a switch on all message types that the protocol may receive. It can then process each message type either inside the switch or by calling a function to handle the message type received.

All event types used by EXata protocols are enumerated in the file EXATA_HOME/include/api.h. If the protocol being added needs additional event types, these should be included in the enumeration in file api.h, as shown in Figure 4-130.

The event dispatcher function for a MAC protocol may handle both packet and timer events. As an example, Figure 4-146 shows the CSMA event dispatcher function MacCsmaLayer. Note that once a message has been processed, it is freed by calling the API MESSAGE_Free. The event dispatcher also includes a default case in the switch statement to handle events of an unknown type.

Note **It is important to free the memory after the message has been processed; otherwise, the simulator will leak memory.**

See files mac_csma.h and mac_csma.cpp for definitions of data structures and functions used for implementing CSMA.

For the CSMA protocol, there is only one type of event: a timer event of type `MSG_MAC_TimerExpred`. When the event occurs, MacCsmaLayer checks if the timeout event corresponds to the latest timer set by the protocol by checking the timer sequence number. If the sequence number does not correspond to the latest timer set by the protocol, MacCsmaLayer ignores the event. If the timer event indicates that the backoff timer has expired, MacCsmaLayer calls function CheckPhyStatusAndSendOrBackoff. If the timer event indicates that the yield timer has expired, MacCsmaLayer calls function MacCsmaPassive.

```
void MacCsmaLayer(Node *node, int interfaceIndex, Message *msg)
{
    /*
     * Retrieve the pointer to the data portion which relates
     * to the CSMA protocol.
     */

    MacDataCsma *csma = (MacDataCsma *)node->macData[interfaceIndex]->macVar;
    int seq_num;
    ...
    assert(msg->eventType == MSG_MAC_TimerExpired);
    seq_num = *((int *) MESSAGE_ReturnInfo(msg);
    MESSAGE_Free(node, msg);
    if ((seq_num < csma->timer.seq) ||
        ((csma->timer.flag & CSMA_TIMER_SWITCH) == CSMA_TIMER_OFF)) {
        return;
    }
    if (seq_num > csma->timer.seq) {
        assert(FALSE);
    }
    assert(((csma->timer.flag & CSMA_TIMER_TYPE) ==
            CSMA_TIMER_BACKOFF) ||
           ((csma->timer.flag & CSMA_TIMER_TYPE) == CSMA_TIMER_YIELD));
    switch(csma->timer.flag & CSMA_TIMER_TYPE) {
    case CSMA_TIMER_BACKOFF:
    {
        csma->timer.flag = CSMA_TIMER_OFF | CSMA_TIMER_UNDEFINED;
        CheckPhyStatusAndSendOrBackoff(node, csma);
        break;
    }
    case CSMA_TIMER_YIELD:
        csma->timer.flag = CSMA_TIMER_OFF | CSMA_TIMER_UNDEFINED;
        csma->status = CSMA_STATUS_PASSIVE;
        MacCsmaPassive(node, csma);
        break;
    default:
        assert(FALSE); abort();
        break;
    }/*switch*/
}
```

FIGURE 4-146. CSMA Event Dispatcher

4.5.6.5 Modifying MAC Layer Functions

This step is similar to the one for adding a wired MAC protocol (see Section 4.5.5.8).

Add code to function MAC_NetworkLayerHasPacketToSend to call MYPROTOCOL's send function when the MAC Layer receives a packet from the Network Layer and MYPROTOCOL is running at the interface (see Figure 4-147). Function MacMyprotocolNetworkLayerHasPacketToSend is the MYPROTOCOL function that handles packets received from the Network Layer. `MacDataMyprotocol` is the data structure for MYPROTOCOL.

```
void
MAC_NetworkLayerHasPacketToSend(Node *node, int interfaceIndex)
{
    /* Select the MAC protocol model, and direct it to send/buffer the
       packet. */
    ...
    switch (node->macData[interfaceIndex]->macProtocol)
    {
        ...
        case MAC_PROTOCOL_DOT11:
        {
            MacDot11NetworkLayerHasPacketToSend(
                node, (MacDataDot11 *) node->macData[interfaceIndex]->macVar);
            break;
        }
        case MAC_PROTOCOL_CSMA:
        {
            MacCsmaNetworkLayerHasPacketToSend(
                node, (MacDataCsma *) node->macData[interfaceIndex]->macVar);
            break;
        }
        case MAC_PROTOCOL_MYPROTOCOL:
        {
            MacMyprotocolNetworkLayerHasPacketToSend(
                node,
               (MacDataMyprotocol *) node->macData[interfaceIndex]->macVar);
            break;
        }
        ...
     }
}
```

FIGURE 4-147. Delivering Packets from Network Layer to MAC Protocols

The MAC Layer also implements several functions that are used by upper layer protocols to determine the type of interface. These functions are listed in Section 4.5.5.8. To add MYPROTOCOL to EXata, modify functions MAC_IsWirelessNetwork and MAC_IsOneHopBroadcastNetwork so that they correctly indicate the type of network that MYPROTCOL runs on. As an example, the modification to function MAC_IsWirelessNetwork is shown in Figure 4-148.

```
BOOL
MAC_IsWirelessNetwork(Node *node, int interfaceIndex)
{
    if (node->macData[interfaceIndex]->macProtocol == MAC_PROTOCOL_802_11 ||
        node->macData[interfaceIndex]->macProtocol == MAC_PROTOCOL_CSMA ||
        node->macData[interfaceIndex]->macProtocol ==
                                                MAC_PROTOCOL_MYPROTOCOL ||
        ...
        node->macData[interfaceIndex]->macProtocol ==
                                            MAC_PROTOCOL_SATELLITE_BENTPIPE)
    {
        return TRUE;
    }
    return FALSE;
}
```

FIGURE 4-148. Determining MAC Protocol Type

A wireless MAC protocol also interacts with the Physical Layer. The Physical Layer calls function MAC_ReceivePacketFromPhy to deliver a packet to the MAC Layer. MAC_ReceivePacketFromPhy calls the receive function for the MAC protocol running at the interface to process the packet received from the Physical Layer. For example, MAC_ReceivePacketFromPhy calls the CSMA receive function MacCsmaReceivePacketFromPhy when CSMA is running at the interface. MAC_ReceivePacketFromPhy is implemented in mac.cpp.

To add a new MAC protocol, MYPROTOCOL, modify MAC_ReceivePacketFromPhy so that the receive function for MYPROTCOL is called when MYPROTOCOL is running at the interface, as shown in Figure 4-149. Function MacMyprotocolReceivePacketFromPhy is the MYPROTCOL function to handle packets received from the Physical Layer.

```
void
MAC_ReceivePacketFromPhy(
    Node *node,
    int interfaceIndex,
    Message *packet)
{
    ...
    if (!MAC_InterfaceIsEnabled(node, interfaceIndex))
    {
        if (node->macData[interfaceIndex]->macProtocol ==
            MAC_PROTOCOL_CELLULAR)
        {
            MESSAGE_FreeList(node, packet);
        }
        else
        {
            MESSAGE_Free(node, packet);
        }
        return;
    }

    switch (node->macData[interfaceIndex]->macProtocol)
    {
        case MAC_PROTOCOL_DOT11: {
            MacDot11ReceivePacketFromPhy(
                node, (MacDataDot11*)node->macData[interfaceIndex]->macVar,
                packet);
            break;
        }
        case MAC_PROTOCOL_CSMA:
        {
            MacCsmaReceivePacketFromPhy(
                     node,
                    (MacDataCsma*)node->macData[interfaceIndex]->macVar,
                    packet);
            break;
        }
        case MAC_PROTOCOL_MYPROTOCOL:
        {
            MacMyprotocolReceivePacketFromPhy(
                     node,
                    (MacDataMyprotocol*)node->macData[interfaceIndex]->macVar,
                    packet);
            break;
        }
        ...
      }
}
```

FIGURE 4-149. Delivering Packets from Physical Layer to MAC Protocols

A wireless MAC protocol also receives and processes notifications of Physical Layer status change. When the status of the Physical Layer changes, the Physical Layer sends a notification to the MAC Layer by using the API MAC_ReceivePhyStatusChangeNotification. MAC_ReceivePhyStatusChangeNotification calls the Physical Layer status change handler function for the MAC protocol running at the interface. For example, if CSMA is running at the interface, MAC_ReceiveStatusChangeNotification calls function

MacCsmaReceivePhyStatusChangeNotification. MAC_ReceivePhyStatusChangeNotification is implemented in mac.cpp.

To add a new MAC protocol, MYPROTOCOL, modify MAC_ReceivePhyStatusChangeNotification so that the Physical Layer status change handler function for MYPROTCOL is called when MYPROTOCOL is running at the interface, as shown in Figure 4-150. Function MacMyprotocolReceivePhyStatusChangeNotification is the MYPROTCOL function to handle Physical Layer status changes.

```
void
MAC_ReceivePhyStatusChangeNotification(
    Node *node,
    int interfaceIndex,
    PhyStatusType oldPhyStatus,
    PhyStatusType newPhyStatus,
    clocktype receiveDuration,
    const Message *potentialIncomingPacket)
{
    switch (node->macData[interfaceIndex]->macProtocol)
    {
        case MAC_PROTOCOL_DOT11:{
            MacDot11ReceivePhyStatusChangeNotification(
                node,
                (MacDataDot11*)node->macData[interfaceIndex]->macVar,
                oldPhyStatus,
                newPhyStatus,
                receiveDuration,
                potentialIncomingPacket);
            break;
        }
        case MAC_PROTOCOL_CSMA:
        {
            MacCsmaReceivePhyStatusChangeNotification(
                node,
                (MacDataCsma*)node->macData[interfaceIndex]->macVar,
                oldPhyStatus,
                newPhyStatus);
            break;
        }
        case MAC_PROTOCOL_MYPROTOCOL:
        {
            MacMyprotocolReceivePhyStatusChangeNotification(
                node,
                (MacDataMyprotocol*)node->macData[interfaceIndex]->macVar,
                oldPhyStatus,
                newPhyStatus);
            break;
        }
        ...
    }
}
```

FIGURE 4-150. Notifying MAC Protocols of Physical Layer Status Changes

4.5.6.6 Interfacing with Network and Physical Layers

In this section we describe the interface between the Network Layer and a wireless MAC Protocol, and the interface between a wireless MAC protocol and the Physical Layer.

A wireless MAC protocol interacts with the Network Layer in the following ways:

1. When IP has a packet to send and the output queue is empty, IP indicates to the MAC protocol that a packet is ready for transmission. If the MAC protocol is in the appropriate state, it dequeues the packet from the output queue. See Section 4.5.6.6.1.
2. When the state of the MAC protocol changes to one where it can transmit a packet, the MAC protocol checks the output queue. If the queue is non-empty, the MAC protocol dequeues a packet from the queue. See Section 4.5.6.6.1.
3. When the MAC protocol receives a packet from the Physical Layer that is meant for the Network Layer, the MAC protocol delivers the packet to the Network Layer. See Section 4.5.6.6.2.
4. When the MAC protocol receives a packet from the Physical Layer that is not addressed to the node, but the node is operating in promiscuous mode, the MAC protocol delivers the packet to the Network Layer. See Section 4.5.6.6.2.
5. Some MAC protocols pass an indication to the Network Layer when certain events occur at the MAC Layer. These events include: a packet being dropped at the MAC Layer and receiving a MAC Layer acknowledgement for a transmitted packet. See Section 4.5.5.9.3.

A wireless MAC protocol interacts with the Physical Layer in the following ways:

1. When the MAC protocol has a packet to send, it checks the status of the Physical Layer to determine if packet transmission can start. See Section 4.5.6.6.1.
2. When the MAC protocol is ready to transmit a packet, it adds a MAC header to the packet and sends the packet to the Physical Layer. See Section 4.5.6.6.1.
3. When the Physical Layer receives a packet from another node, it sends it to the MAC Layer. The MAC Layer removes the MAC header from the received packet and processes the packet. See Section 4.5.6.6.2.
4. When the status of the Physical Layer changes, the Physical Layer notifies the MAC Layer of the status change. The MAC protocol takes appropriate action depending upon the type of status change. See Section 4.5.6.6.3.

4.5.6.6.1 Processing Outgoing Packets

When IP has a packet to send to the MAC Layer and the output queue is empty, IP calls function MAC_NetworkLayerHasPacketToSend. MAC_NetworkLayerHasPacketToSend calls the appropriate function for the MAC protocol running at the interface to process the packet from the Network Layer (see.Section 4.5.6.5). If CSMA is running at the interface, MAC_NetworkLayerHasPacketToSend calls the CSMA function MacCsmaNetworkLayerHasPacketToSend.

MacCsmaNetworkLayerHasPacketToSend checks the status of the node. If the node's status is `CSMA_STATUS_PASSIVE`, MacCsmaNetworkLayerHasPacketToSend calls function CheckPhyStatusAndSendOrBackoff. Function CheckPhyStatusAndSendOrBackoff checks the status of the Physical Layer and of the output queue, and either calls function MacCsmaXmit to transmit a packet or function MacCsmaBackoff to enter backoff state. Functions MacCsmaNetworkLayerHasPacketToSend and CheckPhyStatusAndSendOrBackoff are shown in Figure 4-151 and are implemented in mac_csma.cpp.

```
void MacCsmaNetworkLayerHasPacketToSend(Node *node, MacDataCsma *csma)
{
    if (csma->status == CSMA_STATUS_PASSIVE) {
        CheckPhyStatusAndSendOrBackoff(node, csma);
    }//if//
}


static //inline//
void CheckPhyStatusAndSendOrBackoff(Node* node, MacDataCsma* csma) {
    /* Carrier sense response from phy. */

    if ((PhyStatus(node, csma) == PHY_IDLE) &&
        (csma->status != CSMA_STATUS_IN_XMITING))
    {
        csma->status = CSMA_STATUS_XMIT;
        MacCsmaXmit(node, csma);
    }
    else {
        if (!MAC_OutputQueueIsEmpty(
                node, csma->myMacData->interfaceIndex))
        {
            csma->status = CSMA_STATUS_BACKOFF;
            MacCsmaBackoff(node, csma);
        }
    }
}
```

FIGURE 4-151. Processing Outgoing Packets

As discussed in Section 4.5.6.4.2, CSMA enters the `CSMA_STATUS_PASSIVE` state when the yield timer expires. This indicates that CSMA can transmit a new packet. In this case, MacCsmaLayer calls function MacCsmaPassive. MacCsmaPassive calls function MacCsmaNetworkLayerHasPacketToSend if the output queue is non-empty. As explained before, MacCsmaNetworkLayerHasPacketToSend calls function CheckPhyStatusAndSendOrbackoff, which calls either MacCsmaXmit or MacCsmaBackoff.

Function MacCsmaXmit, shown in Figure 4-152, dequeues a packet from the output queue, adds a header to the packet, and calls function PHY_StartTransmittingSignal to send the packet to the Physical Layer for transmission. Function ConvertVariableHWAddressTo802Address converts the MAC address in the structure MacHWAddress to an Ethernet type address.

```
static
void MacCsmaXmit(Node *node, MacDataCsma *csma)
{
    Message *msg;
    MacHWAddress destHWAddr;
    int networkType;
    TosType priority;
    CsmaHeader         *hdr;

    assert(csma->status == CSMA_STATUS_XMIT);
    /*
     * Dequeue packet which was received from the
     * network layer.
     */
    MAC_OutputQueueDequeuePacket(
        node, csma->myMacData->interfaceIndex,
        &msg, &destHWAddr, &networkType, &priority);
    if (msg == NULL)
    {
        ...
        if(csma->BOtimes >0)
        {
            csma->status = CSMA_STATUS_BACKOFF;
        }
        else
        {
            csma->status = CSMA_STATUS_PASSIVE;
        }
        return;
    }
    csma->status = CSMA_STATUS_IN_XMITING;
    csma->timer.flag = CSMA_TIMER_OFF | CSMA_TIMER_UNDEFINED;

  /* Assign other fields to packet to be sent to phy layer. */
   MESSAGE_AddHeader(node, msg, sizeof(CsmaHeader), TRACE_CSMA);
   hdr = (CsmaHeader *) msg->packet;
   ConvertVariableHWAddressTo802Address(node, &destHWAddr, &hdr->destAddr);
   ConvertVariableHWAddressTo802Address(
                  node,
                  &node->macData[csma->myMacData->interfaceIndex]->macHWAddr,
                  &hdr->sourceAddr);
   hdr->priority = priority;
   PHY_StartTransmittingSignal(node, csma->myMacData->phyNumber,
                               msg, FALSE, 0);
   if (MAC_IsBroadcastMac802Address(&hdr->destAddr)) {
        csma->pktsSentBroadcast++;
   }
   else {
        csma->pktsSentUnicast++;
   }
}
```

FIGURE 4-152. Sending Outgoing Packet to Physical Layer

4.5.6.6.2 Processing Incoming Packets

When the Physical Layer has a packet to send to the MAC Layer, the Physical Layer calls function MAC_ReceivePacketFromPhy. MAC_ReceivePacketFromPhy calls the appropriate function for the MAC protocol running at the interface to process the packet from the Physical Layer (see.Section 4.5.6.5). If CSMA is running at the interface, MAC_ReceivePacketFromPhy calls the CSMA function MacCsmaReceivePacketFromPhy.

MacCsmaReceivePacketFromPhy, shown in Figure 4-153, checks the status of the node. If the node is not in the transmitting state, MacCsmaReceivePacketFromPhy does one of the following:

- If the packet is addressed to the node or is a broadcast packet, MacCsmaReceivePacketFromPhy removes the MAC header and delivers the packet to the Network Layer by calling function MAC_HandoffSucessfullyReceivedPacket.
- If the packet is not addressed to the node and is not a broadcast packet, but the node is operating in promiscuous mode, MacCsmaReceivePacketFromPhy calls function MacCsmaHandlePromiscousMode. MacCsmaHandlePromiscousMode removes the MAC header and sends the packet to the Network Layer by using the function MAC_SneakPeekAtMacPacket.

```
void MacCsmaReceivePacketFromPhy(
    Node* node, MacDataCsma* csma, Message* msg)
{
    if (csma->status == CSMA_STATUS_IN_XMITING) {
        MESSAGE_Free(node, msg);
        return;
    }//if//

    switch (csma->status) {
    case CSMA_STATUS_PASSIVE:
    case CSMA_STATUS_CARRIER_SENSE:
    case CSMA_STATUS_BACKOFF:
    case CSMA_STATUS_YIELD: {
       int interfaceIndex = csma->myMacData->interfaceIndex;
       CsmaHeader *hdr = (CsmaHeader *) msg->packet;
       MacHWAddress  destHWAddress;
       Convert802AddressToVariableHWAddress(node, &destHWAddress,
                                                    &hdr->destAddr);
       if (MAC_IsMyAddress(node, &destHWAddress)) {
            csma->pktsGotUnicast++;
        }
        else if (MAC_IsBroadcastMac802Address(&hdr->destAddr))
        {
            csma->pktsGotBroadcast++;
        }
       if (MAC_IsMyAddress(node, &destHWAddress) ||
                           MAC_IsBroadcastHWAddress(&destHWAddress))
        {
             MacHWAddress srcHWAddress;
             Convert802AddressToVariableHWAddress(node, &srcHWAddress,
                                                    &hdr->sourceAddr);
            MESSAGE_RemoveHeader(node, msg, sizeof(CsmaHeader), TRACE_CSMA);
            MAC_HandOffSuccessfullyReceivedPacket(node,
               csma->myMacData->interfaceIndex, msg, &srcHWAddress);
        }
        else {
            if (node->macData[interfaceIndex]->promiscuousMode) {
                MacCsmaHandlePromiscuousMode(node, csma, msg,
                                        hdr->sourceAddr,  hdr->destAddr);
            }
            MESSAGE_Free(node, msg);
        }
        break;
    }
    default:
        MESSAGE_Free(node, msg);
        printf("MAC_CSMA: Error with node %u, status %ld.\n",
               node->nodeId, csma->status);
        assert(FALSE); abort();
    }//switch//
}
```

FIGURE 4-153. Processing Incoming Packets

Note that besides transporting upper layer packets, some MAC protocols may also generate and receive *control* packets. For example, IEEE 802.11 MAC uses CTS and RTS control packets. These control packets originate at the MAC Layer at the sending node. At the receiving node, the control packets are

processed at the MAC Layer and are not delivered to the upper layers. If MYPROTOCOL uses control packets, the receive function for MYPROTOCOL, MacMyprotocolReceivepacketFromPhy, should check the destination layer of a received packet and should not deliver control packets to the Network Layer.

4.5.6.6.3 Processing Physical Layer Status Change Notification

The operation of a wireless MAC protocol depends upon the state of the Physical Layer. When the status of the Physical Layer changes, the Physical Layer sends a notification to the MAC Layer by using the API MAC_ReceiveStatusChangeNotification. MAC_ReceiveStatusChangeNotification calls the physical status change handler function for the MAC protocol running at the interface (see.Section 4.5.6.5). For example, if CSMA is running at the interface, MAC_ReceiveStatusChangeNotification calls function MacCsmaReceivePhyStatusChangeNotification.

For CSMA, the status change of interest occurs when the Physical Layer status changes from a transmitting state to a non-transmitting state. This status change indicates the end of transmission of a packet by the Physical Layer. When this status change occurs, function MacCsmaReceivePhyStatusChangeNotification, shown in Figure 4-154, resets the backoff parameters. set the CSMA status to `CSMA_STATUS_YIELD`, and calls function MacCsmaYield.

```
void MacCsmaReceivePhyStatusChangeNotification(
   Node* node,
   MacDataCsma* csma,
   PhyStatusType oldPhyStatus,
   PhyStatusType newPhyStatus)
{
   if (oldPhyStatus == PHY_TRANSMITTING) {
      assert(newPhyStatus != PHY_TRANSMITTING);
      assert(csma->status == CSMA_STATUS_IN_XMITING);

      csma->BOmin = CSMA_BO_MIN;
      csma->BOmax = CSMA_BO_MAX;
      csma->BOtimes = 0;
      csma->status = CSMA_STATUS_YIELD;
      MacCsmaYield(node, csma, (clocktype)CSMA_TX_DATA_YIELD_TIME);
   }//if//
}
```

FIGURE 4-154. Processing Physical Layer Status Changes

4.5.6.7 Collecting and Reporting Statistics

This step is similar to the one for adding a wired MAC Protocol (see Section 4.5.5.10).

4.5.6.8 Finalization

This step is similar to the one for adding a wired MAC Protocol (see Section 4.5.5.10.5).

4.5.6.9 Including and Compiling Files

This step is similar to the one for adding a wired MAC Protocol (see Section 4.5.5.12).

4.5.6.10 Integrating the Protocol into the GUI

To make the new protocol available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.6 Physical Layer

The Physical Layer is the lowest layer in the EXata protocol stack (see Figure 4-1). It transmits and receives data over physical media. The Physical Layer interfaces with peer Physical Layer entities in other nodes via the communication medium to provide services to the MAC Layer.

To model the Physical Layer in a simulation we need to incorporate characteristics of the transmitter and the receiver. Modeling the Physical Layer requires modeling all aspects of a wireless system: modulation, coding, noise, interference and antenna gains. In EXata, a Physical Layer model consists of two parts: a PHY component and an antenna component. The PHY component models signal transmission and reception and reflects the effects of the MAC scheme, node status, physical parameters, distortions from the channel, and interference from neighbor nodes. The antenna component models the functions and properties of the antenna.

This section gives a detailed description of how to add a PHY model and an antenna model to EXata. Modeling the communication medium is covered in Section 4.7 and modeling node mobility is covered in Section 4.8.

4.6.1 Physical Layer Models in EXata

EXata provides a number of PHY and antenna models. Table 4-16 lists the different PHY models. Table 4-17 lists the different antenna models. See the corresponding model library for a detailed description of each model and its parameters.

TABLE 4-16. PHY Models in EXata

PHY Model	Description	Model Library
PHY802.11a	Models the IEEE 802.11a PHY specification. This radio operates in the 5 GHz frequency band, uses Orthogonal Frequency Division Multiplexing (OFDM) and supports the following data rates (in Mbits/s): 6, 9, 12, 18, 24, 36, 48, 54.	Wireless
PHY802.11b	Models the IEEE 802.11b PHY specification. This radio operates in the 2.4 GHz frequency band, uses Direct Sequence Spread Spectrum (DSSS) and supports the following data rates (in Mbits/s): 1, 2, 5.5, 11.	Wireless
PHY802.11n	Models the IEEE 802.11n PHY specification. This model improves the network throughput over the IEEE 802.11a and 802.11g standards and achieves a maximum net data rate from 54 to 600 Mbps.	Wireless
PHY802.15.4	Models the IEEE 802.15.4 PHY specification. This radio uses different waveforms in different frequency bands to support different data rates.	Sensor Networks
PHY802.16	Models the IEEE 802.16 PHY specification. This radio uses OFDM and uses the following modulation and encoding combinations: QPSK 1/2, QPSK 3/4, 16QAM 1/2, 16QAM 3/4, 64QAM 1/2, 64QAM 2/3, and 64QAM 3/4.	Advanced Wireless

TABLE 4-16. PHY Models in EXata (Continued)

PHY Model	Description	Model Library
PHY-ABSTRACT	Abstract PHY model. This is a generic PHY model and can be used to simulate different PHYs. This model simulates a PHY that is capable of carrier sensing and is able to work with both BER-based and SNR threshold-based reception models.	Wireless
PHY-GSM	Models the GSM Physical Layer.	Cellular
PHY-LTE	Models the LTE Physical Layer.	LTE
PHY-UMTS	Models the PHY layer for UMTS systems. (To select this protocol, parameter `PHY-MODEL` should be set to `PHY-CELLULAR` and parameter `CELLULAR-PHY-MODEL` should be set to `PHY-UMTS`.)	UMTS

TABLE 4-17. Antenna Models in EXata

Antenna Model	Description	Model Library
OMNIDIRECTIONAL	Omnidirectional antenna model. This is the model for the basic antenna, which yields the same antenna gain irrespective of the signal direction.	Wireless
SWITCHED-BEAM	Switched-beam antenna model. The switched-beam antenna can switch among multiple antenna patterns and uses the pattern that yields the maximum antenna gain.	Wireless
STEERABLE	Steerable antenna model. The steerable antenna can rotate the antenna and uses the direction that yields the maximum antenna gain.	Wireless
PATTERNED	Patterned antenna model. This antenna model uses antenna pattern files in the NSMA and Open ASCII formats in addition to the traditional format.	Wireless

4.6.2 Physical Layer Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to Physical Layer models. These files contain detailed comments on functions and other code components.

The Physical Layer API is composed of several macros, functions, and structures. These are defined in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/phy.h

  This file contains definitions common to Physical Layer models, the Physical Layer data structure in the node structure, and prototypes of functions defined in EXATA_HOME/libraries/wireless/src/phy.cpp.

- EXATA_HOME/include/antenna.h and EXATA_HOME/libraries/wireless/srcantenna_global.h

  These files contain definitions common to antenna models and prototypes of functions defined in antenna.cpp and antenna_global.cpp in EXATA_HOME/libraries/wireless/src, respectively.
- EXATA_HOME/include/mac.h

  This file contains definitions of API functions needed to communicate with the MAC Layer.

Additionally, the following header file is also relevant to the Physical Layer:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.

The following are the folders and source files associated with the Physical Layer:

- EXATA_HOME/libraries/wireless/src

  This folder contains the source and header files for the various Physical Layer models implemented in EXata. The file names are based on the name of the model that they implement, e.g., to see the implementation for IEEE 802.11a, look at files phy_802_11.cpp and phy_802_11.h in this folder.
- EXATA_HOME/libraries/wireless/src/phy.cpp

  This file contains generic Physical Layer functions, including the initialization, message processing and finalization functions.
- antenna.cpp and antenna_global.cpp in EXATA_HOME/libraries/wireless/src

  These files contain implementation of generic antenna functions and the implementation of the omni-directional antenna model.
- EXATA_HOME/libraries/wireless/src/prop_range.cpp

  This file implements the radio-range program, which calculates the likely propagation range of a node, under no interference conditions, using the parameters specified in the configuration file.

4.6.3 Physical Layer Data Structures

The Physical Layer data structures are defined in EXATA_HOME/include/phy.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file phy.h for a complete description.)

1. `PhyModel`: This is an enumeration type that lists all the PHY models.

```
enum PhyModel{
    PHY802_11a,
    PHY802_11b,
    PHY_ABSTRACT,
    PHY_GSM,
    ...
    PHY_NONE
};
```

2. `PhyRxModel`: This is an enumeration type that lists all the reception models. A reception model simulates the scheme used by the PHY model to determine the quality of the received signal.

```
enum PhyRxModel{
    RX_802_11a,
    RX_802_11b,
    RX_802_16,
    RX_UMTS,
    RX_802_15_4,
    SNR_THRESHOLD_BASED,
    BER_BASED,
    PCOM_BASED
};
```

3. `AntennaModel`: This structure holds information about an antenna model. Enumeration types `AntennaModelType` and `AntennaPatternType` are declared in antenna_global.h.

```
struct AntennaModel {
    AntennaModelType   antennaModelType;
    int                numModels;
    AntennaPatternType antennaPatternType;
    void               *antennaVar;
}AntennaModel;
```

4. `PhyData`: This is the main data structure used by the Physical Layer and stores information about the Physical Layer models running at a specific interface. Some important fields of this structure are explained below.

```
struct PhyData {
    int            phyIndex;
    int            macInterfaceIndex;
    Address*       networkAddress;
    D_BOOL*        channelListenable;
    D_BOOL*        channelListening;
    BOOL           phyStats;
    int            channelIndexForTransmission;
    PhyModel       phyModel;
    PhyRxModel     phyRxModel;
    double         phyRxSnrThreshold;
    double         noise_mW_hz;
    int            numBerTables;
    PhyBerTable*   snrBerTables;
    RandomSeed     seed;
    void*          phyVar;
    double         systemLoss_dB;
    AntennaModel*  antennaData;
    BOOL           contentionFreeProp;
    void *         nodeLinkLossList;
    void*          nodeLinkDelayList;
    ...
    double         noiseFactor;
};
```

FIGURE 4-155. `PhyData` Data Structure

- `phyIndex`: This is the Physical Layer index of the interface.
- `macInterfaceIndex`: This is the MAC Layer index of the interface.
- `networkAddress`: This is the network address of the interface.
- `channelListenable`: This is a bit mask that indicates which channels the node can potentially listen to.
- `channelListening`: This is a bit mask that indicates which channels the node is currently listening to.
- `phyStats`: This variable indicates whether statistics collection is enabled for the Physical Layer.
- `channelIndexForTransmission`: This is the index of the channel on which the node is currently transmitting.
- `phyModel`: This variable indicates the PHY model in use at the interface.
- `phyRxModel`: This variable indicates the reception model in use at the interface.
- `phyRxSnrThreshold`: This is the SNR threshold for the interface.
- `noise_mW_hz`: This variable stores the noise floor at the interface.
- `numBerTables`: This variable stores the number of BER tables to be used.
- `snrBerTables`: This is a pointer to the BER tables to be used for determining quality of received signals.
- `seed`: This variable is used to store the seed for the PHY model in use at the interface.
- `phyVar`: This is a pointer to the data structure for the PHY model in use at the interface.
- `systemLoss_db`: This variable stores the total loss in dB which is the sum of connection loss, mismatch loss, cable loss and the loss caused in antenna energy conversion.
- `antennaData`: This is a pointer to the data structure for the antenna model in use at the interface.
- `contentionFreeProp`: This variable indicates whether contention free propagation is enabled.
- `noiseFactor`: This variable stores the noise factor of the interface for the PHY 802.16 model.

4.6.4 Physical Layer APIs and Inter-layer Communication

This section describes the APIs used by the MAC Layer to communicate with the Physical Layer (see Section 4.6.4.1), the APIs used by the Physical Layer to communicate with the MAC Layer (see Section 4.6.4.2), the APIs used by PHY models to communicate with the communication medium (see Section 4.6.4.3), and the APIs used by the communication medium to communicate with PHY models (see Section 4.6.4.4). This section also describes the APIs used by PHY Models to communicate with antenna models (see Section 4.6.4.5) and lists some of the Physical Layer utility APIs (see Section 4.6.4.6).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.6.4.1 MAC Layer to Physical Layer Communication

MAC Layer protocols use several APIs to communicate with the Physical Layer. The prototypes for these API functions are contained in the file phy.h.

Some of the functions used for communication from the MAC Layer to the Physical Layer are listed below.

- PHY_StartTransmittingSignal: This function is used by the MAC Layer to send a packet to the Physical Layer.
- PHY_StartListeningToChannel: This function is used by the MAC Layer to direct the Physical Layer to start listening to the specified channel.
- PHY_StopListeningToChannel: This function is used by the MAC Layer to direct the Physical Layer to stop listening to the specified channel.

- PHY_SetTransmissionChannel: This function is used by the MAC Layer to set the channel for transmission.

4.6.4.2 Physical Layer to MAC Layer Communication

Physical Layer protocols use several APIs to communicate with the MAC Layer. The prototypes for these API functions are contained in the file EXATA_HOME/include/mac.h. The file EXATA_HOME/main/mac.cpp contains the implementation of these functions.

Some of the functions used for communication from the Physical Layer to the MAC Layer are listed below.

- MAC_ReceivePacketFromPhy: This function delivers a packet from the Physical Layer to the MAC Layer.
- MAC_ReceivePhyStatusChangeNotification: This function notifies the MAC Layer of a status change at the Physical Layer.

4.6.4.3 PHY Models to Communication Medium Communication

The communication medium provides the API PROP_ReleaseSignal to enable PHY entities to communicate with the communication medium. A PHY model calls the API PROP_ReleaseSignal to transmit a signal.

The prototype for PROP_ReleaseSignal is contained in the file EXATA_HOME/include/propagation.h.

4.6.4.4 Communication Medium to PHY Models Communication

The communication medium uses the APIs listed below to communicate with PHY models. The prototypes for these functions are contained in phy.h. The file phy.cpp contains the implementation of these functions.

- PHY_SignalArrivalFromChannel: This function indicates the start of a signal.
- PHY_SignalEndFromChannel: This function indicates the end of a signal.

4.6.4.5 PHY Model to Antenna Models Communication

PHY models use several APIs to communicate with antenna models. The prototypes for these functions are contained in the file phy.h. The file phy.cpp contains the implementation of these functions.

Some of the APIs used for communication from PHY models to antenna models are listed below.

- PHY_LockAntennaDirection: This function locks the direction of the antenna.
- PHY_UnlockAntennaDirection: This function unlocks the direction of the antenna.

4.6.4.6 Physical Layer Utility APIs

Several APIs are available at the Physical Layer that perform tasks internal to the Physical Layer. Some of these functions can be used by other layers, as well. Some of the Physical Layer utility APIs are listed below.

The prototypes for the following utility API functions are contained in the file phy.h. The file phy.cpp contains the implementation of these functions.

- PHY_GetTxDataRate: This function returns the transmission data rate.
- PHY_GetRxDataRate: This function returns the reception data rate.
- PHY_SetLowestTxDataRateType: This function sets the lowest transmission data rate type.
- PHY_SetHighestTxDataRateType: This function sets the highest transmission data rate type.
- PHY_GetTransmissionDuration: This function returns the transmission duration of a signal.

The prototypes for the following utility API functions are contained in the file EXATA_HOME/include/antenna.h. The file EXATA_HOME/libraries/wireless/src/antenna.cpp contains the implementation of these functions.

- ANTENNA_IsInOmnidirectionalMode: This function indicates whether the antenna is operating in the omni-directional mode.
- ANTENNA_GainForThisDirection: This function returns the antenna gain for the specified direction.
- ANTENNA_GainForThisSignal: This function returns the antenna gain for the specified signal.

4.6.5 Adding a PHY Model

Although the working of each PHY model is different, there are certain functions that are performed by most PHY models. This section provides an outline for developing and adding a PHY model to EXata. We illustrate the process of adding a PHY model by using as an example the implementation code for the IEEE 802.11a PHY specification. The header file for the IEEE 802.11a implementation is phy_802_11.h and the source file is phy_802_11.cpp in the folder EXATA_HOME/libraries/wireless/src. We use code snippets from these two files throughout this section to illustrate different steps in developing a PHY model. After understanding the discussed snippets, look at the complete code for IEEE 802.11a to understand how a PHY model is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a PHY model, PHY_MYPHY, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.6.6.2).
2. Modify the file phy.cpp to include the model's header file (see Section 4.6.6.2).
3. Include the PHY model in the list of PHY models and the reception model, MYRXMODEL, in the list of reception models (see Section 4.6.6.3).
4. Define data structures for the PHY model (see Section 4.6.6.5).
5. Decide on the format for the PHY model-specific configuration parameters (see Section 4.6.6.6.1).
6. Call the PHY model's initialization function from the Physical Layer initialization function, PHY_CreateAPhyForMac (see Section 4.6.6.6.2).
7. Write the initialization function for the PHY model (see Section 4.6.5.5.3). The initialization function should include the following tasks:
   a. Declare and initialize the state variables.
   b. Read and store the configuration parameters for the PHY model.
   c. Initialize the antenna model.
   d. Set the transmission channel.
8. Call the PHY model's event handler from the Physical Layer event dispatcher, PHY_ProcessEvent (see Section 4.6.5.6).
9. Modify Physical Layer functions to integrate the new PHY model (see Section 4.6.5.7).
10. Write a function to handle outgoing packets (see Section 4.6.5.8.1).
11. Write functions to process the start and end of an incoming packet (see Section 4.6.5.8.2).
12. Include code in various functions to collect statistics.
    a. Declare statistics variables (see Section 4.6.5.9.1).
    b. Initialize the statistics variables in the PHY model's initialization function (see Section 4.6.5.9.2).
    c. Update the statistics as appropriate (see Section 4.6.5.9.3).
    d. Write a function to print the statistics (see Section 4.6.5.9.4).

e. Add dynamic statistics to the protocol, if desired (see Section 4.6.5.9.5) .

13. Call the PHY model finalization function from the Physical Layer finalization function, PHY_Finalize (see Section 4.6.5.10.1).

14. Write the PHY model finalization function (see Section 4.6.5.10.2). Call the function to print statistics from the PHY model finalization function.

15. Modify the file prop_range.cpp to enable the radio-range utility function to calculate the propagation range of a node using the new PHY model (see Section 4.6.5.11).

16. Include the PHY model header and source files in the EXata tree and compile (see Section 4.6.5.12).

17. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.6.5.13).

4.6.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol or model, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new PHY model, the programmer name the different components of the new model in a similar manner. It will be helpful to examine the implementation of the IEEE 802.11a PHY model in EXata for hints for naming and coding different components of the new PHY model.

In this section, we describe the steps for developing a PHY model called “PHY_MYPHY”. We will use the string “PhyMyphy” in the names of the different components of this model, just as the string “Phy802_11” appears in the names of the components of the IEEE 802.11a implementation.

4.6.5.2 Creating Files

The first step towards adding a PHY model is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/wireless/src. However, it is recommended that all user-developed models be made part of a library. In our example, we will place the PHY model in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn’t already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the PHY model and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *phy_* to designate the files as PHY model files.

Examples*:*

- phy_802_11.h, phy_802_11.cpp: These files, in the folder EXATA_HOME/libraries/wireless/src, implement the IEEE 802.11a and IEEE 802.11b PHY models.
- phy_abstract.h, phy_abstract.cpp: These files, in the folder EXATA_HOME/libraries/wireless/src, implement the abstract PHY model.

In keeping with the naming guidelines of Section 4.6.5.1, the header file for the example PHY model is called phy_myphy.h, and the source file is called phy_myphy.cpp.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, phy_myphy.h, should contain the following:

- Constant definitions
- Data structure definitions
- Prototypes for interface functions in the source file, phy_myphy.cpp

The source file, phy_myphy.cpp, should contain the following:

- Statement to include the PHY model's header file:

```
#include "phy_myphy.h"
```

- Statements to include standard library functions and other header files needed by the PHY model's source file. A typical PHY model source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "api.h"
#include "antenna.h"
#include "antenna_global.h"
#include "antenna_switched.h"  //For switched beam antenna model
#include "antenna_steerable.h" //For steerable antenna model
#include "antenna_patterned.h" //For patterned antenna model
```

- Initialization function for the PHY model, PhyMyphyInit
- Finalization function for the PHY model, PhyMyphyFinalize
- PHY model implementation functions

The file EXATA_HOME/libraries/wireless/src/phy.cpp contains the layer level initialization function and functions to implement the PHY model functionality. These layer level functions in turn call the PHY model's initialization, event handler and finalization functions. Therefore, to make these PHY model functions available to the layer level functions, insert the following include statement in the file phy.cpp:

```
#include "phy_myphy.h"
```

4.6.5.3 Including PHY_MYPHY in List of PHY Models

Each node in EXata hosts an operating protocol stack. For each layer in the stack, a list of protocols/ models running at that layer is maintained. When a new PHY model is added to EXata, it needs to be included in the list of PHY models. To do this, add the PHY model's name to the enumeration `PhyModel` defined in phy.h (see Section 4.6.3).

For our example PHY model, add the entry `PHY_MYPHY` to `PhyModel`, as shown in Figure 4-156.

```
enum PhyModel{
    PHY802_11a,
    PHY802_11b,
    PHY_ABSTRACT,
    PHY_GSM,
    ...
    PHY_NONE,
    PHY_MYPHY
};
```

FIGURE 4-156. Adding PHY_MYPHY to List of PHY Models

Note **Always add to the end of lists in header files.**

To add a new reception model, add the entry `RX_MYRXMODEL` to the enumeration `PhyRxModel` defined in phy.h (see Section 4.6.3), as shown in Figure 4-157.

```
enum PhyRxModel{
    RX_802_11a,
    RX_802_11b,
    RX_802_16,
    RX_UMTS,
    RX_802_15_4,
    SNR_THRESHOLD_BASED,
    BER_BASED,
    PCOM_BASED,
    RX_MYRXMODEL
};
```

FIGURE 4-157. Adding MYRXMODEL to List of Reception Models

4.6.5.4 Defining Data Structures

Each PHY model has its own data structures, which are defined in the model's header file. The data structures store information such as:

1. PHY parameters (see Section 4.6.6.6.2)
2. Statistics variables (see Section 4.6.5.9.1)

Define an appropriate data structure, `PhyDataMyphy`, for PHY_MYPHY in the model's header file, phy_myphy.h. As an example, the following data structure, defined in phy_802_11.h, is used by the IEEE 802.11a PHY model:

```
typedef struct struct_phy_802_11_str {
    PhyData*  thisPhy;
    int       txDataRateTypeForBC;
    int       txDataRateType;
    D_Float32 txPower_dBm;
    float     txDefaultPower_dBm[PHY802_11_NUM_DATA_RATES];
    int       rxDataRateType;
    double    rxSensitivity_mW[PHY802_11_NUM_DATA_RATES];
    int       numDataRates;
    int       dataRate[PHY802_11_NUM_DATA_RATES];
    double    numDataBitsPerSymbol[PHY802_11_NUM_DATA_RATES];
    int       lowestDataRateType;
    int       highestDataRateType;
    double    directionalAntennaGain_dB;
    Message*  rxMsg;
    double    rxMsgPower_mW;
    clocktype rxTimeEvaluated;
    BOOL      rxMsgError;
    clocktype rxEndTime;
    Orientation rxDOA;
    Message *txEndTimer;
    D_Int32   channelBandwidth;
    clocktype rxTxTurnaroundTime;
    double    noisePower_mW;
    double    interferencePower_mW;
    PhyStatusType mode;
    PhyStatusType previousMode;
    Phy802_11Stats  stats;
} PhyData802_11;
```

In the above declaration, `Phy802_11Stats` is the statistics data structure for the IEEE 802.11a PHY model. See the declaration of `PhyData802_11` in phy_802_11.h for a description of the fields of the data structure.

4.6.5.5 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a PHY model.

4.6.5.5.1 Determining the PHY Configuration Format

A PHY model may use model-specific configuration parameters for its operation. The configuration parameters are specified in the EXata configuration file. The format for specifying a PHY model's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

Generally, a PHY model requires the transmission power and receiver sensitivity at different transmission rates to be specified. As an example, the following parameters specify a transmission power of 20dBm and a receiver sensitivity of -85.0 dBm for the IEEE 802.11a PHY model when operating at 6Mbps:

```
PHY-MODEL                          PHY802.11a
PHY802.11a-TX-POWER--6MBPS         20.0
PHY802.11a-RX-SENSITIVITY--6MBPS   -85.0
```

Decide on the format for specifying the new PHY model's configuration parameters. For our example PHY model, specify the configuration parameters in the EXata configuration file using the following format (`<Identifier>` and `<Index>` can also be used to qualify the parameter declarations, as described above):

```
PHY-MODEL                  PHY_MYPHY
<param1>                   <value1>
...
<paramN>                   <valueN>
```

where

`<param1>`, ..., `<paramN>`	: Names of parameters for PHY_MYPHY.
`<value1>`, ..., `<valueN>`	: Values of the PHY parameters.

Section 4.6.5.5.3 explains how to read user input specified in this format to initialize the model.

4.6.5.5.2 Calling the PHY Model Initialization Function

The protocol stack of each node is initialized in a bottom up manner. For a wireless network, the MAC Layer and Physical Layer at an interface are initialized together, with the Physical Layer initialization taking

place before the MAC Layer initialization. This process is performed in the node initialization function PARTITION_InitializeNodes, implemented in EXATA_HOME/main/partition.cpp (see Section 3.4.1).

The node initialization function, PARTITION_InitializeNodes, calls the MAC Layer initialization function MAC_Initialize. Function MAC_Initialize reads the configuration file for lines starting with the keywords `SUBNET` or `LINK`. If the input line begins with the keyword `SUBNET`, MAC_Initialize calls the function ProcessInputFileSubnetLine. If the input line begins with the keyword `LINK`, MAC_Initialize calls the function ProcessInputFileLinkLine. Function ProcessInputFileSubnetLine assigns an IP address to the subnet interface for each node in the subnet and calls function AddNodeToSubnet for each node. Function AddNodeToSubnet initializes the interface information for the subnet interface. Functions MAC_Initialize, ProcessInputFileSubnetLine, ProcessInputFileLinkLine, and AddNodeToSubnet are implemented in the file EXATA_HOME/main/mac.cpp.

For a wireless MAC protocol, function AddNodeToSubnet initializes the Physical Layer model specified for the interface by calling the function PHY_CreateAPhyForMac. For example, if IEEE 802.11a is specified as the PHY model running at the interface, AddNodeToSubnet calls PHY_CreateAPhyForMac with `PHY802_11a` as the `PhyModel` parameter. Modify AddNodeToSubnet to call PHY_CreateAPhyForMac with `PHY_MYPHY` as the `PhyModel` parameter if PHY_MYPHY is specified as the PHY model for the interface, as shown in Figure 4-158. Function PHY_CreateAPhyForMac is implemented in phy.cpp.

```
static void //inline//
AddNodeToSubnet(
    Node *node,
    const NodeInput *nodeInput,
    ...
    int         subnetListIndex)
{
    int interfaceIndex;
    ...
    IO_ReadString(
        node,
        node->nodeId,
        interfaceIndex,
        nodeInput,
        "PHY-MODEL",
        &phyModelFound,
        phyModelName);
    ...
    if (strncmp(macProtocolName, "FCSC-", 5) == 0) {
    ...
    }
    else {
        PhyModel phyModel = PHY802_11b;
        ...
        if (strncmp(phyModelName, "FCSC-", 5) == 0) {
            ...
        }
        else if (strcmp(phyModelName, "PHY802.11a") == 0) {
            PHY_CreateAPhyForMac(
                    node,
                    nodeInput,
                    interfaceIndex,
                    &address,
                    PHY802_11a,
                    &node->macData[interfaceIndex]->phyNumber);
            phyModel = PHY802_11a;
        }
        else if (strcmp(phyModelName, "PHY_MYPHY") == 0) {
            PHY_CreateAPhyForMac(
                    node,
                    nodeInput,
                    interfaceIndex,
                    &address,
                    PHY_MYPHY,
                    &node->macData[interfaceIndex]->phyNumber);
            phyModel = PHY_MYPHY;
        }
        ...
}
```

FIGURE 4-158. Calling the Physical Layer Initialization Function

Function PHY_CreateAPhyForMac, shown in Figure 4-159, performs the following tasks:

- Reads and stores the listenable and listening channel masks from the configuration file.
- Reads and stores the generic Physical Layer parameters from the configuration file.
- Reads the reception model to be used for the interface and sets the receiver parameters (SNR threshold or BER tables) accordingly. For example, if the reception model is specified to be IEEE 802.11a, PHY_CreateAPhyForMac calls the IEEE 802.11a function Phy802_11aSetBerTable to set up the BER tables.
- Calls the initialization function for the PHY model running at the interface. For example, if IEEE 802.11a is specified as the PHY model running at the interface, PHY_CreateAPhyForMac calls the IEEE 802.11a initialization function Phy802_11Init, which is implemented in phy_802_11.cpp.

To add your PHY model to EXata, make the following modifications to PHY_CreateAPhyForMac, as shown in Figure 4-159:

- Call the function MyrxmodelSetBerTable to set up the BER tables according to the desired reception scheme, if MYRXMODEL is specified as the reception model to be used at the interface.
- Call the PHY_MYPHY initialization function, PhyMyphyInit, if PHY_MYPHY is specified as the PHY model for the interface.

The prototypes for the functions MyrxmodelSetBerTable and PhyMyphyInit should be included in the header file, phy_myphy.h.

```
void PHY_CreateAPhyForMac(Node *node, const NodeInput *nodeInput,
                          int interfaceIndex, Address *networkAddress,
                          PhyModel phyModel, int* phyNumber)
{
    char buf[10*MAX_STRING_LENGTH];
    ...
    int phyIndex = node->numberPhys;
    PhyData *thisPhy;
    ...
    thisPhy = (PhyData *)MEM_malloc(sizeof(PhyData));
    memset(thisPhy, 0, sizeof(PhyData));
    node->phyData[phyIndex] = thisPhy;
    ...
    thisPhy->phyModel = phyModel;
    assert(phyModel == PHY802_11a ||
           phyModel == PHY802_11b ||
           phyModel == PHY_MYPHY ||
           ...
           phyModel == PHY802_15_4);
    ...
    // Set PHY-RX-MODEL
     IO_ReadString(node, node->nodeId, interfaceIndex, nodeInput,
                   "PHY-RX-MODEL", &wasFound, buf);
    if (wasFound) {
        if (strcmp(buf, "PHY802.11a") == 0) {
            thisPhy->phyRxModel = RX_802_11a;
            Phy802_11aSetBerTable(thisPhy);
        }
        else if (strcmp(buf, "MYRXMODEL") == 0) {
            thisPhy->phyRxModel = RX_MYRXMODEL;
            MyrxmodelSetBerTable(thisPhy);
        }
        else
        ...
    }
    else {
        ...
    }
    ...
    switch(thisPhy->phyModel) {
        case PHY802_11b:
        case PHY802_11a: {
            Phy802_11Init(node, phyIndex, nodeInput);
            break;
        }
        case PHY_MYPHY: {
            PhyMyphyInit(node, phyIndex, nodeInput);
            break;
        }
        ...
    }/*switch*/
} //PHY_CreateAPhyForMacLayer//
```

FIGURE 4-159. Calling the PHY Model Initialization Function

4.6.5.5.3 Implementing the PHY Model Initialization Function

The initialization of a PHY model takes place in the initialization function of the model that is called by the Physical Layer initialization function PHY_CreateAPhyForMac. The initialization function of a PHY model commonly performs the following tasks:

- Create an instance of the PHY model data structure
- Read and store the PHY model's parameters
- Initialize the state variables of the PHY model
- Initialize the antenna model
- Set the transmission channel

The initialization function initializes the PHY model state. Each PHY model has a structure that it uses to store state information. This may include information such as the model state and parameters, statistics variables, etc. Each instance of the PHY model maintains its own state variable.

To store the state, declare the structure to hold the PHY model's state in the header file, phy_myphy.h (see Section 4.6.5.4). As an example, see the declaration of the IEEE 802.11a data structure `PhyData802_11` in phy_802_11.h.

Create an instance of the PHY model state by allocating memory to the state structure. IEEE 802.11a performs this task in its initialization function Phy802_11Init by calling the function MEM_malloc to allocate memory for the IEEE 802.11a data structure `PhyData802_11`, as shown in Figure 4-160. Phy802_11Init and the other IEEE 802.11a functions are implemented in phy_802_11.cpp. Data structure and constant definitions for IEEE 802.11a are contained in phy_802_11.h.

Phy802_11Init also sets up pointers between the newly created instance of the IEEE 802.11a data structure `PhyData802_11` and the data structure that stores the Physical Layer information for the interface, `phyData[phyIndex]`.

Next, Phy802_11Init initializes the antenna model by calling function ANTENNA_Init. Function ANTENNA_Init is implemented in EXATA_HOME/libraries/wireless/src/antenna.cpp.

The initialization function of a PHY model also stores the values of parameters that it requires in its operation. These parameters may be default parameters or user-specified configuration parameters. Phy802_11Init sets the default parameters for IEEE 802.11a by calling function Phy802_11aInitializeDefaultParameters. Some configurable parameters are read within Phy802_11Init and the others are read by calling function Phy802_11aSetUserConfigurableParameters.

The configurable parameters are read using IO functions such as IO_ReadBool, IO_ReadInt and IO_ReadDouble to read parameter values from the input file and set the appropriate fields of the PHY model data structure `PhyData802_11`. IO_ReadBool, IO_ReadInt, IO_ReadDouble and other IO functions are defined in EXATA_HOME/include/fileio.h.

The state variables for the PHY model are also initialized in the initialization function. For example, Phy802_11Init initializes the PHY model status, message buffer, etc.

The initialization function also initializes the channels on which the PHY model can transmit by calling function PHY_SetTransmissionChannel, which is defined in phy.h.

```
void Phy802_11Init(
    Node *node,
    const int phyIndex,
    const NodeInput *nodeInput)
{
    BOOL   wasFound;
    ...
    PhyData802_11 *phy802_11 =
        (PhyData802_11 *)MEM_malloc(sizeof(PhyData802_11));
    memset(phy802_11, 0, sizeof (PhyData802_11));
    node->phyData[phyIndex]->phyVar = (void*)phy802_11;
    phy802_11->thisPhy = node->phyData[phyIndex];
    ...
    // Antenna model initialization
    //
    ANTENNA_Init(node, phyIndex, nodeInput);
    ...
    if (node->phyData[phyIndex]->phyModel == PHY802_11a) {
        Phy802_11aInitializeDefaultParameters(node, phyIndex);
        Phy802_11aSetUserConfigurableParameters(node, phyIndex, nodeInput);
    }
    else if (node->phyData[phyIndex]->phyModel == PHY802_11b) {
        ...
    }
    ...
    IO_ReadBool(
        node->nodeId,
        node->phyData[phyIndex]->networkAddress,
        nodeInput,
        "PHY802.11-AUTO-RATE-FALLBACK",
        &wasFound,
        &yes);
   ...
    // Initialize status of phy
    //
    phy802_11->rxMsg = NULL;
    ...
    phy802_11->mode = PHY_IDLE;
    Phy802_11ChangeState(node, phyIndex, PHY_IDLE);
    //
    // Setting up the channel to use for both TX and RX
    //
    for (i = 0; i < numChannels; i++) {
        if (phy802_11->thisPhy->channelListening[i] == TRUE) {
            break;
        }
    }
    assert(i != numChannels);
    PHY_SetTransmissionChannel(node, phyIndex, i);
    return;
}
```

FIGURE 4-160. IEEE 802.11a Initialization Function

4.6.5.6 Implementing the Event Handler

In this section, we describe the steps for implementing the event handler function for a PHY model.

As explained in Section 3.4.2, when an event occurs, it is first handled by the node level dispatcher function NODE_ProcessEvent, defined in EXATA_HOME/main/node.cpp. If the event is for the Physical Layer, NODE_ProcessEvent calls the Physical Layer event dispatcher PHY_ProcessEvent, defined in phy.cpp.

Function PHY_ProcessEvent implements the Physical Layer event dispatcher that informs the appropriate PHY model of received events. Messages contain the index of the interface for which the event has occurred. The API function MESSAGE_GetInstanceId returns the interface index. PHY_ProcessEvent implements a switch statement on the PHY model that is running at the interface read from the message and calls the appropriate model-specific event handler. For example, if IEEE 802.11a is running at the interface, PHY_ProcessEvent calls the IEEE 802.11a event handler function, Phy802_11TransmissionEnd, which is implemented in phy_802_11.cpp.

For the Physical Layer, there is only one event of interest, `MSG_PHY_TransmissionEnd`, which is a timer event. `MSG_PHY_TransmissionEnd` indicates the end of transmission of a packet by a node. To enable the PHY model PHY_MYPHY to process events, add code to PHY_ProcessEvent to call PHY_MYPHY's event handler function when messages for PHY_MYPHY are received. Figure 4-161 shows a code fragment from PHY_ProcessEvent with sample code for calling PHY_MYPHY's event handler function PhyMyphyTransmissionEnd.

Write the event handler function PhyMyphyTransmissionEnd to take appropriate actions when the event `MSG_PHY_TransmissionEnd` occurs. Include the prototype for PhyMyphyTransmissionEnd in the header file, phy_myphy.h.

```
void PHY_ProcessEvent(Node *node, Message *msg) {
    int phyIndex = MESSAGE_GetInstanceId(msg);

   ...
   switch(node->phyData[phyIndex]->phyModel) {
        case PHY802_11b:
        case PHY802_11a: {
            switch (msg->eventType) {
                case MSG_PHY_TransmissionEnd: {
                    Phy802_11TransmissionEnd(node, phyIndex);
                    MESSAGE_Free(node, msg);
                    break;
                }
                default: abort();
            }
            break;
        }
        case PHY_MYPHY: {
            switch (msg->eventType) {
                case MSG_PHY_TransmissionEnd: {
                    PhyMyphyTransmissionEnd(node, phyIndex);
                    MESSAGE_Free(node, msg);
                    break;
                }
                 default: abort();
            }
            break;
        }
        ...
   }
}
```

FIGURE 4-161. Physical Layer Event Dispatcher

4.6.5.7 Modifying Generic Physical Layer Functions

The Physical Layer implements several generic functions that are called by MAC protocols or by communication media models. These generic functions, in turn, call the function for the PHY model that is running at the interface. For example, a MAC protocol sends a packet to the Physical Layer by calling function PHY_StartTransmittingSignal. PHY_StartTransmittingSignal, in turn, calls the function for the PHY model running at the interface. If IEEE 802.11a is running at the interface, PHY_StartTransmittingSignal calls function Phy802_11StartTransmittingSignal.

To add the new PHY model, PHY_MYPHY, to EXata, these generic Physical Layer functions should be modified so that the appropriate PHY_MYPHY function is called when PHY_MYPHY is running at the interface. As an example, Figure 4-162 shows the modifications required for function PHY_StartTransmitingSignal, where PhyMyphyStartTransmittingSignal is the MYPHY function that transmits a packet received from the MAC Layer.

```
void PHY_StartTransmittingSignal(
    Node *node,
    int phyNum,
    Message *msg,
    BOOL useMacLayerSpecifiedDelay,
    clocktype delayUntilAirborne,
    NodeAddress destAddr)
{
    ...
    switch(node->phyData[phyNum]->phyModel) {
        case PHY802_11b:
        case PHY802_11a: {
            Phy802_11StartTransmittingSignal(
                node, phyNum, msg,
                useMacLayerSpecifiedDelay, delayUntilAirborne);
            break;
        }
        case PHY_MYPHY: {
            PhyMyphyStartTransmittingSignal(
                node, phyNum, msg,
                useMacLayerSpecifiedDelay, delayUntilAirborne);
            break;
        }
        ...
    }
}
```

FIGURE 4-162. Modifying a Generic Physical Layer Function

The generic Physical Layer functions that need to be modified are listed below. Depending upon the functionality of the new PHY model being added, not all these functions may need to be modified, or additional functions may need to be written.

1. PHY_StartTransmittingSignal
2. PHY_GetStatus
3. PHY_SignalArrivalFromChannel
4. PHY_SignalEndFromChannel
5. PHY_GetTxDataRate
6. PHY_GetRxDataRate
7. PHY_SetTxDataRateType
8. PHY_GetRxDataRateType
9. PHY_GetTxDataRateType
10. PHY_SetLowestTxDataRateType
11. PHY_GetLowestTxDataRateType
12. PHY_SetHighestTxDataRateType
13. PHY_GetHighestTxDataRateType
14. PHY_SetHighestTxDataRateTypeForBC
15. PHY_GetHighestTxDataRateTypeForBC
16. PHY_GetTransmissionDuration

17. PHY_SetTransmitPower
18. PHY_GetTransmitPower
19. PHY_GetLastSignalsAngleOfArrival
20. PHY_TerminateCurrentReceive
21. PHY_StartTransmittingSignalDirectionally
22. PHY_LockAntennaDirection
23. PHY_UnlockAntennaDirection
24. PHY_MediumIsIdle
25. PHY_MediumIsIdleInDirection
26. PHY_SetSensingDirection
27. PHY_PropagationRange

The modifications to these functions are similar to the modifications shown in Figure 4-162.

4.6.5.8 Interfacing with MAC Layer and Communication Medium

A PHY model interacts with a wireless MAC protocol in the following ways:

1. When the MAC protocol is ready to transmit a packet, it sends the packet to the Physical Layer. See Section 4.6.5.8.1.
2. When the Physical Layer receives a packet from another node, it sends it to the MAC Layer. See Section 4.6.5.8.2.
3. When the status of the Physical Layer changes, the Physical Layer notifies the MAC Layer of the status change. See Section 4.6.5.8.2.
4. The Physical Layer implements several utility functions for use by MAC protocols to perform various tasks, such as locking or unlocking the antenna, setting and retrieving data rates, getting the transmission duration, etc. See Section 4.6.5.7.

A PHY model interacts with a communication medium model in the following ways:

1. When the PHY model has a packet to send, it adds a Physical Layer header and sends the packet to the communication medium. See Section 4.6.5.8.1.
2. The communication medium indicates to the PHY model the beginning and end of a transmission from another node. See Section 4.6.5.8.2.

4.6.5.8.1 Processing Outgoing Packets

When a MAC protocol has a packet to send to the Physical Layer, the MAC protocol calls function PHY_StartTransmittingSignal. PHY_StartTransmittingSignal calls the transmit function of the PHY model running at the interface to process the packet from the MAC Layer. For example, if IEEE 802.11a is running at the interface, PHY_StartTransmittingSignal calls the IEEE 802.11a function Phy802_11StartTransmittingSignal (see.Section 4.6.5.7).

Phy802_11StartTransmittingSignal calls the IEEE 802.11a function StartTransmittingSignal to transmit a packet. StartTransmittingSignal and the other IEEE 802.11a functions are implemented in phy_802_11.cpp. StartTransmittingSignal performs the following tasks (see Figure 4-163 and Figure 4-164):

- StartTransmittingSignal calls function PHY_GetTransmissionChannel to get the index of the channel on which to transmit the signal. PHY_GetTransmissionChannel is defined in phy.h.
- If PHY is currently receiving a signal, i.e., the status of PHY is `PHY_RECEIVING`, the PHY model updates the interference power, and resets the receive parameters by calling Phy802_11UnlockSignal.
- StartTransmittingSignal changes the status of PHY to `PHY_TRANSMITTING`.
- StartTransmittingSignal calculates the transmission duration of the packet by calling Phy802_11GetFrameDuration, and adds a Physical Layer header to the packet by calling MESSAGE_AddHeader.
- StartTransmittingSignal calls function PHY_StopListeningToChannel to stop receiving on the channel.
- StartTransmittingSignal calls the communication medium function PROP_ReleaseSignal to transmit the packet. PROP_ReleaseSignal is defined in EXATA_HOME/include/propagation.h.
- StartTransmittingSignal schedules a self-timer of type `MSG_PHY_TransmissionEnd` to indicate the end of transmission of the packet.

```
static
void StartTransmittingSignal(
    Node* node,
    int phyIndex,
    Message* packet,
    BOOL useMacLayerSpecifiedDelay,
    clocktype initDelayUntilAirborne,
    BOOL sendDirectionally,
    double azimuthAngle)
{
    ...
    clocktype delayUntilAirborne = initDelayUntilAirborne;
    PhyData* thisPhy = node->phyData[phyIndex];
    PhyData802_11* phy802_11 = (PhyData802_11 *)thisPhy->phyVar;
    int channelIndex;
    Message *endMsg;
    int packetsize = MESSAGE_ReturnPacketSize(packet);
    clocktype duration;

    PHY_GetTransmissionChannel(node, phyIndex, &channelIndex);
    ...
    if (phy802_11->mode == PHY_RECEIVING) {
        if (thisPhy->antennaModel == ANTENNA_OMNIDIRECTIONAL) {
            phy802_11->interferencePower_mW += phy802_11->rxMsgPower_mW;
        }
        else {
            if (!sendDirectionally) {
               ANTENNA_SetToDefaultMode(node, phyIndex);
            }//if//
            ...
            PHY_SignalInterference(
                node,
                phyIndex,
                channelIndex,
                NULL,
                NULL,
                &(phy802_11->interferencePower_mW));
        }
        Phy802_11UnlockSignal(phy802_11);
    }
    Phy802_11ChangeState(node, phyIndex, PHY_TRANSMITTING);
    ...
}
```

FIGURE 4-163. Processing Outgoing Packets: Calculating Interference Power

```
static
void StartTransmittingSignal(
    Node* node,
    ...
    BOOL sendDirectionally,
    double azimuthAngle)
{
    ...
    Phy802_11ChangeState(node, phyIndex, PHY_TRANSMITTING);
    duration =
        Phy802_11GetFrameDuration(
            thisPhy, phy802_11->txDataRateType, packetsize);
    MESSAGE_AddHeader(node, packet, sizeof(Phy802_11PlcpHeader),
                      TRACE_802_11);
    char* plcpl = MESSAGE_ReturnPacket(packet);
    memcpy(plcpl, &phy802_11->txDataRateType, sizeof(int));
    ...
    PHY_StopListeningToChannel(node, phyIndex, channelIndex);
    ...
    if (AntennaIsInOmnidirectionalMode(node, phyIndex)) {
        PROP_ReleaseSignal(
            node,
            packet,
            phyIndex,
            channelIndex,
            phy802_11->txPower_dBm,
            duration,
            delayUntilAirborne);
    } else {
        PROP_ReleaseSignal(
            node,
            packet,
            phyIndex,
            channelIndex,
            (float)(phy802_11->txPower_dBm -
             phy802_11->directionalAntennaGain_dB),
            duration,
            delayUntilAirborne);
    }//if//
    ...
    endMsg = MESSAGE_Alloc(node,
                           PHY_LAYER,
                           0,
                           MSG_PHY_TransmissionEnd);
    MESSAGE_SetInstanceId(endMsg, (short) phyIndex);
    MESSAGE_Send(node, endMsg, delayUntilAirborne + duration + 1);
    ...
}
```

FIGURE 4-164. Processing Outgoing Packets: Sending Packet to Communication Medium

4.6.5.8.2 Processing Incoming Packets

When the PHY model at a node transmits a packet, it calls the communication medium function PROP_ReleaseSignal (see Section 4.6.5.8.1). Based on the relative positions of the nodes and the transmission parameters, such as transmit power, antenna gain, and data rate, the communication medium determines which nodes can receive the signal. For each of the neighbor nodes that can receive the signal transmitted by a node, the communication medium makes two function calls: function PHY_SignalArrivalFromChannel to indicate the start of a packet, and function PHY_SignalEndFromChannel to indicate the end of a packet.

PHY_SignalArrivalFromChannel and PHY_SignalEndFromChannel call the functions for the PHY model running at the interface (see Section 4.6.5.7). For example, if IEEE 802.11a is running at the interface, PHY_SignalArrivalFromChannel calls the function Phy802_11SignalArrivalFromChannel, and PHY_SignalEndFromChannel calls the function Phy802_11SignalEndFromChannel.

Phy802_11SignalArrivalFromChannel, shown in Figure 4-165 and Figure 4-166, performs the following tasks:

- If the PHY model status is `PHY_RECEIVING`, i.e., the node is already receiving another signal, Phy802_11SignalArrivalFromChannel calculates the receive power and determines if there are any errors in the portion of the packet received so far by calling function Phy802_11CheckRxPacketError. Phy802_11SignalArrivalFromChannel then adds the receive power to the interference power.
- If the PHY model status is `PHY_IDLE` or `PHY_SENSING`, Phy802_11SignalArrivalFromChannel calculates the interference power and received power.
- If the received power is greater than the receiver sensitivity, Phy802_11SignalArrivalFromChannel locks on to the signal by calling Phy802_11LockSignal, changes status to `PHY_RECEIVING`, and informs the MAC Layer of the status change by calling Phy802_11ReportExtendedStatusToMac.
- If the received power is less than the receiver sensitivity, Phy802_11SignalArrivalFromChannel calls function Phy802_11CarrierSensing to determine if the signal strength is high enough to trigger a status change. If a status change is triggered, Phy802_11SignalArrivalFromChannel updates the status and informs the MAC Layer of the status change by calling Phy802_11ReportStatusToMac.

```
void Phy802_11SignalArrivalFromChannel(
    Node* node,
    int phyIndex,
    int channelIndex,
    PropRxInfo *propRxInfo)
{
    PhyData *thisPhy = node->phyData[phyIndex];
    PhyData802_11* phy802_11 = (PhyData802_11*) thisPhy->phyVar;
    assert(phy802_11->mode != PHY_TRANSMITTING);
    ...
    switch (phy802_11->mode) {
        case PHY_RECEIVING: {
            double rxPower_mW =
                NON_DB(ANTENNA_GainForThisSignal(node, phyIndex, propRxInfo) +
                       propRxInfo->rxPower_dBm);

            if (!phy802_11->rxMsgError) {
               phy802_11->rxMsgError =
                  Phy802_11CheckRxPacketError(node, phy802_11, NULL);
            }//if//

            phy802_11->rxTimeEvaluated = getSimTime(node);
            phy802_11->interferencePower_mW += rxPower_mW;

            break;
        }
        case PHY_IDLE:
        case PHY_SENSING:
        {
           ...
        }
        default:
            abort();
         }//switch (phy802_11->mode)//
}
```

FIGURE 4-165. Processing Start of Incoming Signal in `PHY_RECEIVING` Mode

```
void Phy802_11SignalArrivalFromChannel(...)
{
    ...
    switch (phy802_11->mode) {
        ...
        case PHY_IDLE:
        case PHY_SENSING:
        {
            double rxInterferencePower_mW = NON_DB(
                ANTENNA_GainForThisSignal(node, phyIndex, propRxInfo) +
                propRxInfo->rxPower_dBm);
            double rxPowerInOmni_mW = NON_DB(
                ANTENNA_DefaultGainForThisSignal(node, phyIndex, propRxInfo) +
                propRxInfo->rxPower_dBm);

            if (rxPowerInOmni_mW >= phy802_11->rxSensitivity_mW[0]) {
                PropTxInfo *propTxInfo
                    = (PropTxInfo *)MESSAGE_ReturnInfo(propRxInfo->txMsg);
                ...
                if (!AntennaIsLocked(node, phyIndex)) {
                   ANTENNA_SetToBestGainConfigurationForThisSignal(
                       node, phyIndex, propRxInfo);
                   PHY_SignalInterference(...);
                }
                else {
                    rxPower_mW = rxInterferencePower_mW;
                }
                Phy802_11LockSignal(...);
                Phy802_11ChangeState(node, phyIndex, PHY_RECEIVING);
                Phy802_11ReportExtendedStatusToMac(...);
            }
            else {
                PhyStatusType newMode;
                phy802_11->interferencePower_mW += rxInterferencePower_mW;
                if (Phy802_11CarrierSensing(node, phy802_11)) {
                   newMode = PHY_SENSING;
                } else {
                   newMode = PHY_IDLE;
                }//if//
                if (newMode != phy802_11->mode) {
                    Phy802_11ChangeState(node, phyIndex, newMode);
                    Phy802_11ReportStatusToMac(node, phyIndex, newMode);
                }//if//
            }//if//
            break;
        }
        ...
    }//switch (phy802_11->mode)//
}
```

FIGURE 4-166. Processing Start of Incoming Signal in `PHY_IDLE` and `PHY_SENSING` Modes

Phy802_11SignalEndFromChannel, shown in Figure 4-167 and Figure 4-168, performs the following tasks:

- Phy802_11SignalEndFromChannel checks if there are any errors in the received packet by calling Phy802_11CheckRxPacketError.

- If the PHY model status is PHY_RECEIVING and the received signal is the one that the PHY model had locked on to, Phy802_11SignalEndFromChannel stops receiving the signal and calls Phy802_11UnlockSignal.
- Phy802_11SignalEndFromChannel calls Phy802_11CarrierSensing and changes the PHY model status to PHY_SENSING or PHY_IDLE depending on the interference power.
- If the packet was received without any errors, Phy802_11SignalEndFromChannel removes the Physical Layer header and sends the packet to the MAC Layer by calling MAC_ReceicePacketFromPhy. MAC_ReceicePacketFromPhy is implemented in EXATA_HOME/main/mac.cpp.
- If the packet was received with errors, Phy802_11SignalEndFromChannel reports the status change to the MAC Layer by calling Phy802_11ReportStatusToMac and drops the packet.
- If the PHY model status is not PHY_RECEIVING or the received signal is not the one that the PHY model had locked on to, Phy802_11SignalEndFromChannel updates the interference power. If the PHY model status is not PHY_RECEIVING, Phy802_11SignalEndFromChannel calls Phy802_11CarrierSensing and changes the PHY model status to PHY_SENSING or PHY_IDLE depending on the interference power.
- If the PHY model status changes, Phy802_11SignalEndFromChannel reports the status change to the MAC Layer by calling Phy802_11ReportStatusToMac.

```
void Phy802_11SignalEndFromChannel(
    Node* node,
    int phyIndex,
    int channelIndex,
    PropRxInfo *propRxInfo)
{
    PhyData *thisPhy = node->phyData[phyIndex];
    PhyData802_11* phy802_11 = (PhyData802_11*) thisPhy->phyVar;
    double sinr = -1.0;
    BOOL receiveErrorOccurred = FALSE;
    ...
    assert(phy802_11->mode != PHY_TRANSMITTING);
    if (phy802_11->mode == PHY_RECEIVING) {
        if (phy802_11->rxMsgError == FALSE) {
            phy802_11->rxMsgError =
                Phy802_11CheckRxPacketError(node, phy802_11, &sinr);
            phy802_11->rxTimeEvaluated = getSimTime(node);
        }//if
    }//if//
   receiveErrorOccurred = phy802_11->rxMsgError;
    // If the phy is still receiving this signal, forward the frame
    // to the MAC layer.
    if ((phy802_11->mode == PHY_RECEIVING) &&
        (phy802_11->rxMsg == propRxInfo->txMsg))
    {
        ...
    }
    else {
        PhyStatusType newMode;
        double rxPower_mW =
            NON_DB(ANTENNA_GainForThisSignal(node, phyIndex, propRxInfo) +
                   propRxInfo->rxPower_dBm);
        phy802_11->interferencePower_mW -= rxPower_mW;
        if (phy802_11->interferencePower_mW < 0.0) {
            phy802_11->interferencePower_mW = 0.0;
        }
        if (phy802_11->mode != PHY_RECEIVING) {
           if (Phy802_11CarrierSensing(node, phy802_11) == TRUE) {
               newMode = PHY_SENSING;
           } else {
               newMode = PHY_IDLE;
           }//if//
           if (newMode != phy802_11->mode) {
               Phy802_11ChangeState(node, phyIndex, newMode);
               Phy802_11ReportStatusToMac(
                   node,
                   phyIndex,
                   newMode);
           }//if//
        }//if//
    }//if//
}
```

FIGURE 4-167. Processing End of Incoming Signal in Non-receiving Mode

```
void Phy802_11SignalEndFromChannel(...)
{
    ...
    assert(phy802_11->mode != PHY_TRANSMITTING);
    if (phy802_11->mode == PHY_RECEIVING) {
        if (phy802_11->rxMsgError == FALSE) {
            phy802_11->rxMsgError =
                Phy802_11CheckRxPacketError(node, phy802_11, &sinr);
            phy802_11->rxTimeEvaluated = getSimTime(node);
        }//if
    }//if//
    receiveErrorOccurred = phy802_11->rxMsgError;
    if ((phy802_11->mode == PHY_RECEIVING) &&
        (phy802_11->rxMsg == propRxInfo->txMsg))
    {
        Message *newMsg;
        if (!ANTENNA_IsLocked(node, phyIndex)) {
            ANTENNA_SetToDefaultMode(node, phyIndex);
            ...
            PHY_SignalInterference(node, phyIndex, channelIndex, NULL,
                                   NULL, &(phy802_11->interferencePower_mW));
        }//if//
        ...
        Phy802_11UnlockSignal(phy802_11);
        if (Phy802_11CarrierSensing(node, phy802_11) == TRUE) {
            Phy802_11ChangeState(node, phyIndex, PHY_SENSING);
        }
        else {
            Phy802_11ChangeState(node, phyIndex, PHY_IDLE);
        }
        if (!receiveErrorOccurred) {
            newMsg = MESSAGE_Duplicate(node, propRxInfo->txMsg);
            MESSAGE_RemoveHeader(
                node, newMsg, sizeof(Phy802_11PlcpHeader), TRACE_802_11);
            ...
            MESSAGE_SetInstanceId(newMsg, (short) phyIndex);
            MAC_ReceivePacketFromPhy(node,
                                     node->phyData[phyIndex]->macInterfaceIndex,
                                       newMsg);
            phy802_11->stats.totalRxSignalsToMac++;
        }
        else {
            Phy802_11ReportStatusToMac(node, phyIndex, phy802_11->mode);
            phy802_11->stats.totalSignalsWithErrors++;
        }//if//
    }
    else {
            ...
    }//if//
}
```

FIGURE 4-168. Processing End of Incoming Signal in `PHY_RECEIVING` Mode

4.6.5.9 Collecting and Reporting Statistics

In this section, we describe how to collect and report statistics for a PHY model.

4.6.5.9.1 Declaring Statistics Variables

A PHY model can be configured to record statistics specified by the programmer, such as:

- Number of signals transmitted
- Number of signals received with errors
- Number of signals received without errors

To enable statistics collection for the PHY model, include the statistic collection variables in the structure used to hold the PHY model state (see Section 4.6.5.4). The statistics related variables can also be defined in a structure and then that structure is included in the state variable. For example, the data structure for IEEE 802.11a, `PhyData802_11`, contains the IEEE 802.11a statistics variable, `Phy802_3Stats`, shown below:

```
typedef struct phy_802_11_stats_str {
    D_Int32 totalTxSignals;
    D_Int32 totalRxSignalsToMac;
    D_Int32 totalSignalsLocked;
    D_Int32 totalSignalsWithErrors;
    D_Float64 energyConsumed;
    D_Clocktype turnOnTime;
} Phy802_11Stats;
```

`PhyData802_11` and `Phy802_11Stats` are defined in phy_802_11.h.

4.6.5.9.2 Initializing Statistics

Initialize statistics variables in the PHY model's initialization function. For example, the IEEE 802.11a initialization function Phy802_11Init, shown in Figure 4-169, initializes all fields of the statistics variable `Phy802_11Stats` to 0.

```
void Phy802_11Init(
    Node *node,
    const int phyIndex,
    const NodeInput *nodeInput)
{
    BOOL   wasFound;
    ...
    //
    // Initialize phy statistics variables
    //
    phy802_11->stats.totalRxSignalsToMac = 0;
    phy802_11->stats.totalSignalsLocked = 0;
    phy802_11->stats.totalSignalsWithErrors = 0;
    phy802_11->stats.totalTxSignals = 0;
    phy802_11->stats.energyConsumed = 0.0;
    phy802_11->stats.turnOnTime = getSimTime(node);
    ...
}
```

FIGURE 4-169. Initializing Statistics Variables for IEEE 802.11a

4.6.5.9.3 Updating Statistics

After declaring and initializing the statistics variables, update their value during the execution of the PHY model, as required. For example, IEEE 802.11a increments the value of `totalRxSignalsToMac` in function Phy802_11SignalEndFromChannel (implemented in phy_802_11.cpp) every time IEEE 802.11a sends a received packet to the MAC Layer, as shown in Figure 4-168.

4.6.5.9.4 Printing Statistics

As a final step towards statistics collection, create a function to print statistics. Call this function from the finalization function of the PHY model, which is discussed in Section 4.6.5.10.2. Alternatively, the statistics can be printed from the finalization function directly.

4.6.5.9.5 Adding Dynamic Statistics

Dynamic statistics are statistic variables whose values can be observed in the EXata GUI during the simulation. See Section 5.2.3 for adding dynamic statistics to a protocol. Refer to *EXata User's Guide* for details of viewing dynamic statistics during the simulation.

4.6.5.10 Finalization

The finalization function of the PHY model is called by the simulator at the end of simulation. It is the last code that executes during the simulation. This function is responsible for printing statistics to the statistics file.

At the end of simulation, the finalization function for each model is called to print the model statistics. As discussed in Section 3.4.3, the finalization function is called hierarchically. The node finalization function, PARTITION_Finalize, which is defined in EXATA_HOME/main/partition.cpp, calls the finalization function for the Physical Layer, PHY_Finalize, defined in phy.cpp. PHY_Finalize calls the finalization function of the PHY model running at each interface.

4.6.5.10.1 Modifying the Physical Layer Finalization Function

Call the finalization function of the PHY model from the Physical Layer finalization function, PHY_Finalize, defined in phy.cpp. Figure 4-170 shows the outline of code that needs to be added to PHY_Finalize. Function PhyMyphyFinalize is the finalization function of the PHY model PHY_MYPHY (see Section 4.6.5.10.2).

```
void PHY_Finalize(Node *node) {
    int phyNum;

    for (phyNum = 0; (phyNum < node->numberPhys); phyNum++) {
        ...
        switch(node->phyData[phyNum]->phyModel) {
            case PHY802_11b:
            case PHY802_11a: {
                Phy802_11Finalize(node, phyNum);

                break;
            }
            case PHY_MYPHY:
            {
                PhyMyphyFinalize(node, phyNum);
                break;
            }
            ...
        }
    }
}
```

FIGURE 4-170. Physical Layer Finalization Function

4.6.5.10.2 Implementing the PHY Model Finalization Function

Write the finalization function for the PHY model PHY_MYPHY, PhyMyphyFinalize. If statistics collection is enabled for the Physical Layer, call the function to print the PHY model's statistics (see Section 4.6.5.9.4) from the finalization function, or add code directly to PhyMyphyFinalize to print statistics. Use the IEEE 802.11a finalization function, Phy802_11Finalize, shown in Figure 4-171, as a template. Phy802_11Finalize is implemented in phy_802_11.cpp.

Function Phy802_11Finalize calls the C function sprintf to create a single string containing the statistic name and statistic value, and then calls function IO_PrintStat to print that string to a file. Function IO_PrintStat function, defined in EXATA_HOME/include/fileio.h, requires the following parameters:

- Node pointer: Pointer to the node reporting the statistics.
- Layer: String indicating the layer. Set this to `"Physical"` for the Physical Layer.
- Protocol: String indicating the model name.
- Interface address: Interface address. Set this to `ANY_DEST` for PHY models.
- Instance identifier: Physical channel index.
- Buffer: String containing the statistics.

```
void Phy802_11Finalize(Node *node, const int phyIndex) {
    PhyData* thisPhy = node->phyData[phyIndex];
    PhyData802_11* phy802_11 = (PhyData802_11*) thisPhy->phyVar;
    char buf[MAX_STRING_LENGTH];

    if (thisPhy->phyStats == FALSE) {
        return;
    }
    assert(thisPhy->phyStats == TRUE);
    sprintf(buf, "Signals transmitted = %d",
            (int) phy802_11->stats.totalTxSignals);
    IO_PrintStat(node, "Physical", "802.11", ANY_DEST, phyIndex, buf);
    sprintf(buf, "Signals received and forwarded to MAC = %d",
            (int) phy802_11->stats.totalRxSignalsToMac);
    IO_PrintStat(node, "Physical", "802.11", ANY_DEST, phyIndex, buf);
...
}
```

FIGURE 4-171. Finalization Function for IEEE 802.11a

As for all other functions, specify the prototype of the finalization function in the PHY model's header file, phy_myphy.h.

4.6.5.11 Modifying Radio-range Utility Function

The file EXATA_HOME/libraries/wireless/src/prop_range.cpp implements the radio-range program, which calculates the likely propagation range of a node, under no interference conditions, using the parameters specified in the configuration file. Modify this file, as shown in Figure 4-172, to incorporate PHY_MYPHY.

```
/*
 * Calculates prop range
 */
...
#include "phy_abstract.h"
#include "phy_myphy.h"
#include "propagation.h"
...

int main(int argc, char **argv) {
    NodeInput       nodeInput;
    int             numNodes = 0;
    ...
    PHY_Init(node, &nodeInput);
    PHY_GlobalBerInit(&nodeInput);

    {
        int interfaceIndex;
        ...
        IO_ReadString(node->nodeId, &networkAddress, &nodeInput,
                      "PHY-MODEL", &found, phyModelName);

        assert(found == TRUE);

        if (strcmp(phyModelName, "PHY802.11a") == 0) {
            PHY_CreateAPhyForMac(node, &nodeInput, interfaceIndex,
                                 &networkAddress, PHY802_11a,
                                 &node->macData[interfaceIndex]->phyNumber);

            phyModel = PHY802_11a;
        }
        else
           if (strcmp(phyModelName, "PHY_MYPHY") == 0) {
            PHY_CreateAPhyForMac(node, &nodeInput, interfaceIndex,
                                 &networkAddress, PHY_MYPHY,
                                 &node->macData[interfaceIndex]->phyNumber);

            phyModel = PHY_MYPHY;
        }
        else
         ...
        else {
            ERROR_ReportError("Unknown PHY-MODEL");
        }
    }
    PROP_Init(node, 0, &nodeInput);
    propProfile = node->partitionData->propChannel[0].profile;
    thisRadio = node->phyData[radioNumber];
    distance = PHY_PropagationRange(node, radioNumber, TRUE);
    return 0;
}
```

FIGURE 4-172. Modifying Radio-range Utility Function

4.6.5.12 Including and Compiling Files

The final step in integrating your PHY model into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the PHY model in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Wireless library, then modify EXATA_HOME/libraries/wireless/Makefile-common as shown in Figure 4-173. Recompile EXata after making the changes.

```
...
# common sources
#
WIRELESS_SRCS = \
$(WIRELESS_DIR)/antenna.cpp \
$(WIRELESS_DIR)/antenna_global.cpp \
...
$(WIRELESS_DIR)/phy_802_11.cpp \
$(WIRELESS_DIR)/phy_abstract.cpp \
$(WIRELESS_DIR)/phy_cellular.cpp \
$(WIRELESS_DIR)/phy_myphy.cpp \
$(WIRELESS_DIR)/propagation.cpp \
$(WIRELESS_DIR)/prop_itm.cpp \
$(WIRELESS_DIR)/prop_plmatrix.cpp \
$(WIRELESS_DIR)/routing_aodv.cpp \
...
```

FIGURE 4-173. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.6.5.13 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.6.6 Adding an Antenna Model

Although the working of each antenna is different, there are certain functions that are performed by most antenna models. This section provides an overview of the flow of an antenna model and provides an outline for developing and adding an antenna model, MYANTENNA, to EXata. The new antenna model may use a new antenna pattern type, MYPATTERN.

The following list summarizes the actions that need to be performed for adding an antenna model, MYANTENNA, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.6.6.2).
2. Modify the files antenna.cpp and antenna_global.cpp to include the antenna model's header file (see Section 4.6.6.2).
3. Modify the file phy.cpp and the source files for any PHY models that use the new antenna model to include the antenna model's header file (see Section 4.6.6.2).
4. Include the antenna model in the list of antennas (see Section 4.6.6.3).

5. If the antenna model uses a new antenna pattern type, include it in the list of antenna pattern types (see Section 4.6.6.4).
6. Define data structures for the antenna model (see Section 4.6.6.5).
7. Decide on the format for the antenna model-specific configuration parameters (see Section 4.6.6.6.1).
8. Call the antenna model's initialization function from the antenna initialization function, ANTENNA_Init (see Section 4.6.6.6.2).
9. Modify function ANTENNA_GlobalAntennaModelInit to read the antenna model's configuration parameters (see Section 4.6.6.6.3).
10. If the antenna model uses antenna pattern files of a new type, then modify function ANTENNA_GlobalAntennaPatternInit to read pattern files of the new type (see Section 4.6.6.6.4).
11. Write the initialization function for the antenna model (see Section 4.6.6.6.5).
12. Modify the generic antenna functions to integrate the new antenna model (see Section 4.6.6.7).
13. Write functions to implement the antenna model functionality (see Section 4.6.6.8).
14. Modify Physical Layer and PHY model functions to integrate the new antenna model (see Section 4.6.6.9).
15. Include the antenna model header and source files in the EXata tree and compile (see Section 4.6.6.10).
16. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.6.6.11).

4.6.6.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the model, the layer in which the model resides, and the functionality of the component, as appropriate. We recommend that when adding a new antenna model, the programmer name the different components of the new model in a similar manner. It will be helpful to examine the implementation of the patterned antenna model in EXata for hints for naming and coding different components of the new antenna model.

In this section, we describe the steps for developing an antenna model called "MYANTENNA". We will use the string "Myantenna" in the names of the different components of this model, just as the string "AntennaPatterned" appears in the names of the components of the patterned antenna implementation.

4.6.6.2 Creating Files

This step is similar to the one for PHY models (see Figure 4.6.5.2). Create the header and source files for the antenna model. Name these files in a way that clearly indicates the model that they implement. For antenna models, prefix the file names with *antenna_*.

Examples*:*

- antenna_steerable.h, antenna_steerable.cpp: These files, in the directory EXATA_HOME/libraries/wireless/src, implement the steerable antenna model.
- antenna_patterened.h, antenna_patterened.cpp: These files, in the directory EXATA_HOME/libraries/wireless/src, implement the patterned antenna model.

In keeping with the naming guidelines of Section 4.6.6.1, the header file for the example antenna model is called antenna_myantenna.h, and the source file is called antenna_myantenna.cpp.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, antenna_myantenna.h, should contain the following:

- Prototypes for interface functions in the source file, antenna_myantenna.cpp
- Constant definitions
- Data structure definitions

The source file, antenna_myantenna.cpp, should contain the following:

- Statement to include the antenna model's header file:

```
#include "antenna_myantenna.h"
```

- Statements to include standard library functions and other header files needed by the antenna model's source file. A typical antenna source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "partition.h"        // EXATA_HOME/main/partition.h
#include "api.h"              // EXATA_HOME/include/api.h
#include "antenna.h"
                    // EXATA_HOME/libraries/wireless/src/antenna.h
#include "antenna_global.h"
                // EXATA_HOME/libraries/wireless/src/antenna_global.h
```

- Initialization function for the antenna model, MyantennaInit
- Antenna model implementation functions

The files antenna.cpp and antenna_global.cpp in the folder EXATA_HOME/libraries/wireless/src contain the layer level initialization function and functions to implement the antenna functionality. These layer level functions in turn call the antenna model's initialization and implementation functions. Therefore, to make these antenna model functions available to the layer level functions, insert the following include statement in the files antenna.cpp and antenna_global.cpp:

```
#include "antenna_myantenna.h"
```

This include statement should also be included in the file EXATA_HOME/libraries/wireless/src/phy.cpp and the source files for any PHY models that use the antenna model.

4.6.6.3 Including MYANTENNA in List of Antenna Models

When a new antenna model is added to EXata, it needs to be included in the list of antenna models. To do this, add the antenna model's name to the enumeration `AntennaModelType` defined in EXATA_HOME/libraries/wireless/src/antenna_global.h.

For our example model, add the entry `ANTENNA_MYANTENNA` to `AntennaModelType`, as shown in Figure 4-174.

```
enum AntennaModelType {
    ANTENNA_OMNIDIRECTIONAL,
    ANTENNA_SWITCHED_BEAM,
    ANTENNA_STEERABLE,
    ANTENNA_PATTERNED,
    ANTENNA_MYANTENNA
};
```

FIGURE 4-174. Adding MYANTENNA to List of Antenna Models

Note **Always add to the end of lists in header files.**

4.6.6.4 Including MYPATTERN in List of Antenna Pattern Types

If the new antenna model uses a new antenna pattern type, it needs to be included in the list of antenna pattern types. To do this, add the antenna pattern type's name to the enumeration `AntennaPatternType` defined in antenna_global.h.

For our example model, the pattern type used is called MYPATTERN. Add the entry `ANTENNA_MYPATTERN` to `AntennaPatternType`, as shown in Figure 4-175.

```
enum AntennaPatternType {
    ANTENNA_PATTERN_TRADITIONAL,
    ANTENNA_PATTERN_ASCII2D,
    ANTENNA_PATTERN_ASCII3D,
    ANTENNA_PATTERN_NSMA,
    ANTENNA_PATTERN_EBE,
    ANTENNA_PATTERN_ASAPS,
    ANTENNA_MYPATTERN
};
```

FIGURE 4-175. Adding MYPATTERN to List of Antenna Pattern Types

4.6.6.5 Defining Data Structures

Each antenna model has its own data structure, which is defined in the antenna model's header file. The data structure stores antenna model-specific information.

Define an appropriate data structure, `AntennaMyantenna`, for MYANTENNA in the antenna model's header file, antenna_myantenna.h. As an example, the following data structure, defined in EXATA_HOME/libraries/wireless/src/antenna_patterned.h, is used by the patterned antenna:

```
typedef struct struct_Antenna_Patterned {
    int                     modelIndex;
    int                     numPatterns;
    int                     patternIndex;
    float                   antennaHeight;
    float                   antennaGain_dB;
    AntennaPattern          *pattern;
} AntennaPatterned;
```

4.6.6.6 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of an antenna model.

4.6.6.6.1 Determining the Configuration Format for Input Parameters

An antenna model may use model-specific configuration parameters for its operation. Configuration parameters for antenna models can be specified in the configuration file, e.g., default.config, or in an antenna models configuration file, e.g., default.antenna-models. It is recommended that the configuration parameters for the new antenna model, MYANTENNA, be specified in the antenna models configuration file. The format for specifying an antenna model's configuration parameters in the antenna models configuration file is:

```
<Parameter-name> [<Index>] <Parameter-value>
```

where

`<Parameter-name>` : Name of the parameter.

`<Index>` : Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.

`<Parameter-value>` : Value to be used for the parameter.

For example, the following lines from the file default.antenna-models show a part of the specification of an antenna model, `DEFAULT1-STEERABLE`, which is a steerable antenna with the specified parameters:

```
ANTENNA-MODEL-NAME      DEFAULT1-STEERABLE
ANTENNA-MODEL-TYPE      STEERABLE
ANTENNA-MODEL-CLASS     DYNAMIC
ANTENNA-GAIN            0.0
ANTENNA-HEIGHT          1.5
ANTENNA-EFFICIENCY      0.8
...
```

The following line from the configuration file, defualt.config, specifies that `DEFAULT1-STEERABLE` is the antenna model to be used for node 18:

```
[18] ANTENNA-MODEL DEFAULT1-STEERABLE
```

Decide on the format for specifying the new antenna model's configuration parameters. Section 4.6.6.6.3 explains how to read user input specified in this format. As an example, the following specification defines an antenna model, `DEFAULT1-MYANTENNA`, which is an antenna of type MYANTENNA, and has the

parameter values listed. This antenna model uses an antenna pattern of type MYPATTERN (see Section 4.6.6.4).

```
ANTENNA-MODEL-NAME       DEFAULT1-MYANTENNA
ANTENNA-MODEL-TYPE       MYANTENNA
ANTENNA-MODEL-CLASS      DYNAMIC
ANTENNA-GAIN             0.0
ANTENNA-HEIGHT           1.5
ANTENNA-EFFICIENCY       0.8
ANTENNA-MISMATCH-LOSS    0.3
ANTENNA-CABLE-LOSS       0.0
ANTENNA-CONNECTION-LOSS 0.2
ANTENNA-PATTERN-NAME     PATTERN-MYPATTERN
ANTENNA-PATTERN-NUM-PATTERNS     1
ANTENNA-PATTERN-TYPE     MYPATTERN
ANTENNA-PATTERN-PATTERN-FILE     default.mypattern
```

4.6.6.6.2 Calling the Antenna Model Initialization Function

The initialization function of an antenna model is called from the layer level antenna initialization function, ANTENNA_Init, implemented in EXATA_HOME/libraries/wireless/src/antenna.cpp. ANTENNA_Init is called by the initialization function of the PHY model running at the interface. For example, if IEEE 802.11a is running at an interface, the PHY model is initialized by calling the IEEE 802.11a initialization function Phy802_11Init (see Section 4.6.5.5.3), which in turn calls function ANTENNA_Init.

Function ANTENNA_Init reads the name of the antenna model specified for the interface, and calls the function ANTENNA_InitFromConfigFile if an omni-directional, steerable, or switched-beam antenna model is specified for the interface. (Configuration parameters for these three antenna models can be specified in the main configuration file as well as the antenna models configuration file.) If a different name is specified for the antenna model, ANTENNA_Init calls function ANTENNA_GlobalAntennaModelGet. If the antenna model has already been encountered before and has an entry in the global antenna structure, ANTENNA_GlobalAntennaModelGet returns a pointer to the structure for the antenna model. If the antenna model has not been encountered before, i.e., ANTENNA_GlobalAntennaModelGet returns a null pointer, ANTENNA_Init calls functions ANTENNA_MakeAntennaModelInput and ANTENNA_GlobalAntennaModelInit to create and initialize an entry for the antenna model in the global antenna structure. ANTENNA_GlobalAntennaModelInit also reads and stores the configuration parameters for the antenna model.

Function ANTENNA_MakeAntennaModelInput is implemented in antenna.cpp. Functions ANTENNA_GlobalAntennaModelGet and ANTENNA_GlobalAntennaModelInit are implemented in EXATA_HOME/libraries/wireless/src/antenna_global.cpp. Function IO_ReadString reads the name of the antenna model from the configuration file. The prototype for IO_ReadString is defined in EXATA_HOME/include/fileio.h

ANTENNA_Init then calls the initialization function for the antenna model. Figure 4-176 shows the modifications that need to be made to ANTENNA_Init to incorporate MYANTENNA in EXata. ANTENNA_MyantennaInit is the initialization function for MYANTENNA.

```
void ANTENNA_Init( Node* node, int phyIndex, const NodeInput* nodeInput)
   {
    PhyData* phyData = node->phyData[phyIndex];
    ...

    IO_ReadString(node->nodeId, phyData->networkAddress, nodeInput,
                  "ANTENNA-MODEL", &wasFound, buf);
    if (!wasFound || (strcmp(buf, "OMNIDIRECTIONAL") == 0)
        || (strcmp(buf, "SWITCHED-BEAM") == 0)
        || (strcmp(buf, "STEERABLE") == 0))
    {
        ANTENNA_InitFromConfigFile(node, phyIndex,nodeInput);
    }
    else
    {
        antennaModel
            = ANTENNA_GlobalAntennaModelGet(node->partitionData, buf);
        // Create a new global antenna structure
        if (antennaModel == NULL)
        {
            NodeInput* antennaModelInput
                = ANTENNA_MakeAntennaModelInput(nodeInput, buf);
            ...
            ANTENNA_GlobalAntennaModelInit(node,
                phyIndex ,antennaModelInput, buf);

            antennaModel
                = ANTENNA_GlobalAntennaModelGet(node->partitionData, buf);
            ...
        }
        ...
        if (antennaModel->antennaModelType == ANTENNA_OMNIDIRECTIONAL)
        {
            ANTENNA_OmniDirectionalInit(node, phyIndex, antennaModel);
            return;
        }
        ...
        else if (antennaModel->antennaModelType == ANTENNA_PATTERNED)
        {
            ANTENNA_PatternedInit(node, phyIndex, antennaModel);
            return;
        }
        else if (antennaModel->antennaModelType == ANTENNA_MYANTENNA)
        {
            ANTENNA_MyantennaInit(node, phyIndex, antennaModel);
            return;
        }
        else
        ...
    }//end of else
}
```

FIGURE 4-176. Calling Antenna Model Initialization Function from ANTENNA_Init

4.6.6.6.3 Reading Configuration Parameters

Function ANTENNA_GlobalAntennaModelInit reads the configuration parameters associated with an antenna model from the antenna models configuration file and stores them in the global antenna data structure. Parameters for each antenna model are stored in a data structure of type `AntennaModelGlobal`, which is defined in antenna_global.h.

In addition to ANTENNA_Init, function ANTENNA_GlobalAntennaModelInit also needs to be modified to incorporate MYANTENNA in EXata. ANTENNA_GlobalAntennaModelInit is implemented in antennna_global.cpp. Figure 4-177 shows the modifications needed to add the antenna model, MYANTENNA. If MYANTENNA uses any additional configuration parameters (see Section 4.6.6.6.1), then add appropriate fields to the data structure `AntennaModelGlobal` and modify ANTENNA_GlobalAntennaModelInit to read those parameters and store them in `AntennaModelGlobal`.

```
void ANTENNA_GlobalAntennaModelInit(
     Node* node, int  phyIndex, const NodeInput* antennaModelInput,
     const char* antennaModelName)
{
    char    buf[MAX_STRING_LENGTH];
    BOOL    wasFound;
    ...
    // Get new model
    AntennaModelGlobal* antennaModel =
        &node->partitionData->antennaModels[
                                      node->partitionData->numAntennaModels];

    // Read in antenna model
    // Model name initialization with
    // ANTENNA-MODEL (required)

    strcpy(antennaModel->antennaModelName, antennaModelName);
    IO_ReadString(node->nodeId, phyData->networkAddress, antennaModelInput,
                  "ANTENNA-MODEL-TYPE", &wasFound, buf);
    ...
    if (strcmp(buf, "OMNIDIRECTIONAL") == 0)
    {
        antennaModel->antennaModelType = ANTENNA_OMNIDIRECTIONAL;
    }
    else if (strcmp(buf, "MYANTENNA") == 0)
    {
        antennaModel->antennaModelType = ANTENNA_MYANTENNA;
    }
    ...
    IO_ReadFloat(node->nodeId, phyData->networkAddress, antennaModelInput,
                 "ANTENNA-HEIGHT", &wasFound,  &height);
    if (wasFound)
    {
        ERROR_Assert(height >= 0 ,  "Illegal height given in the file.\n");
         antennaModel->height = (float) height;
    }
    else
    {
        antennaModel->height = ANTENNA_DEFAULT_HEIGHT;
    }
    ...
    if (antennaModel->antennaModelType != ANTENNA_OMNIDIRECTIONAL)
    {
        antennaModel->antennaPatterns =
            ANTENNA_GlobalModelAssignPattern(node, phyIndex,
                                             antennaModelInput, antennaModel);
    }
    node->partitionData->numAntennaModels++;
    return;
}
```

FIGURE 4-177. Modifications to Function ANTENNA_GlobalAntennaModelInit

4.6.6.6.4 Reading Antenna Pattern Files

If the antenna model is a directional antenna, ANTENNA_GlobalAntennaModelInit calls function ANTENNA_GlobalModelAssignPattern (see Figure 4-177) to associate the proper antenna pattern type with the antenna model. ANTENNA_GlobalModelAssignPattern reads the antenna pattern name associated with the antenna model. If the antenna pattern name has not been encountered before, ANTENNA_GlobalModelAssignPattern calls function ANTENNA_GlobalAntennaPatternInit to initialize the structure associated with the antenna pattern type. ANTENNA_GlobalModelAssignPattern and ANTENNA_GlobalAntennaPatternInit are implemented in antenna_global.cpp.

If the new antenna model uses a new antenna pattern type, then ANTENNA_GlobalAntennaPatternInit should be modified to read pattern files of the new type. Figure 4-178 shows the modification required to ANTENNA_GlobalAntennaPatternInit to read an antenna pattern file of type MYPATTERN where ANTENNA_ReturnMypatternPatternFile is the function to read a pattern file of type MYPATTERN and store the pattern data in the antenna data structure.

If the antenna model uses a new antenna pattern type, write the function ANTENNA_ReturnMypatternPatternFile. Like all other functions belonging to the antenna model, the prototype for ANTENNA_ReturnMypatternPatternFile should be included in the antenna's header file, antenna_myantenna.h.

```
void ANTENNA_GlobalAntennaPatternInit(
     Node* node,
     int phyIndex,
     const NodeInput* antennaModelInput,
     AntennaModelGlobal* antennaModel,
     const char* antennaPatternName)
{
    char    buf[MAX_STRING_LENGTH];
    BOOL    wasFound;

    PhyData *phyData = node->phyData[phyIndex];

    ...
    AntennaPattern* antennaPatterns =
        &node->partitionData->antennaPatterns[
                                    node->partitionData->numAntennaPatterns];
    strcpy(antennaPatterns->antennaPatternName , antennaPatternName);

    IO_ReadString(
        node->nodeId,
        phyData->networkAddress,
        antennaModelInput,
        "ANTENNA-PATTERN-TYPE",
        &wasFound,
        buf);
    ...
    // Assign pattern
    if (strcmp(buf, "ASCII2D") == 0)
    {
        antennaPatterns->antennaPatternType = ANTENNA_PATTERN_ASCII2D;
        ANTENNA_ReturnAsciiPatternFile(node, phyIndex,antennaModelInput,
            antennaPatterns);
    }

    else if (strcmp(buf, "MYPATTERN") == 0)
    {
        antennaPatterns->antennaPatternType = ANTENNA_MYPATTERN;
        ANTENNA_ReturnMypatternPatternFile(node, phyIndex,antennaModelInput,
            antennaPatterns);
    }
    ...

    node->partitionData->numAntennaPatterns++;
    return;
}
```

FIGURE 4-178. Reading Antenna Pattern Files

4.6.6.6.5 Implementing the Antenna Model Initialization Function

The initialization of an antenna model takes place in the initialization function of the antenna that is called by the layer-level antenna initialization function ANTENNA_Init (see Figure 4-176). The initialization function of an antenna model commonly performs the following tasks:

- Create an instance of the antenna model data structure
- Copies the antenna parameters in the antenna model data structure from the global antenna structure

As an example, Figure 4-179 shows the initialization function for the patterned antenna model, ANTENNA_PatternedInit. ANTENNA_PatternedInit performs the following tasks:

- ANTENNA_PatternedInit creates an instance of the patterned antenna data structure, `AntennaPatterned`, by calling function AntennaPatternedAlloc.
- ANTENNA_PatternedInit initializes the fields of the patterned antenna data structure with the values read into the global antenna structure (see Section 4.6.6.6.3).
- ANTENNA_PatternedInit creates an instance of the generic antenna data structure, `AntennaModel`, and initializes the `antennaData` field of the Physical Layer data structure for that interface to point to the newly created generic antenna data structure.
- ANTENNA_PatternedInit initializes the fields of the generic antenna data structure with the appropriate antenna model type and antenna pattern type, and makes the `antennaVar` field of the generic antenna data structure point to the newly created instance of the patterned antenna data structure, `AntennaPatterned`.

ANTENNA_PatternedInit and AntennaPatternedAlloc are implemented in antenna_patterned.cpp.

Write the initialization function for MYANTENNA, ANTENNA_MyantennaInit, to perform similar tasks for MYANTENNA. Like all other functions belonging to the antenna model, the prototype for the initialization function, ANTENNA_MyantennaInit, should be included in the antenna's header file, antenna_myantenna.h.

```
void ANTENNA_PatternedInit(
     Node* node,
     int  phyIndex,
     const  AntennaModelGlobal* antennaModel)

{
    PhyData* phyData = node->phyData[phyIndex];

    phyData->antennaData =
        (AntennaModel*) MEM_malloc(sizeof(AntennaModel));

    ERROR_Assert(phyData->antennaData ,
        "memory allocation problem for phyData->antennaData.\n");
    memset(phyData->antennaData, 0,sizeof(AntennaModel));

    AntennaPatterned *antennaVars = AntennaPatternedAlloc();

    antennaVars->patternIndex = ANTENNA_PATTERN_NOT_SET;


    // init structure
    antennaVars->modelIndex = 0;
    antennaVars->numPatterns = antennaModel->antennaPatterns->numOfPatterns;
    antennaVars->antennaHeight = antennaModel->height;
    antennaVars->antennaGain_dB = antennaModel->antennaGain_dB;
    antennaVars->patternIndex = ANTENNA_DEFAULT_PATTERN;
    antennaVars->pattern = antennaModel->antennaPatterns;

    // Assign antenna model based on Node's model type

    phyData->antennaData->antennaVar = antennaVars;
    phyData->antennaData->antennaModelType = antennaModel->antennaModelType;
    phyData->antennaData->numModels++;
    phyData->antennaData->antennaPatternType =
    antennaModel->antennaPatterns->antennaPatternType;
    phyData->systemLoss_dB = antennaModel->systemLoss_dB;
}
```

FIGURE 4-179. Antenna Model Initialization Function

4.6.6.7 Modifying Generic Antenna Functions

An antenna model implements several functions that are used by PHY models. These antenna model functions are called indirectly by PHY models. For example, if IEEE 802.11a is used as the PHY model, and the PHY needs to lock the antenna in the direction of maximum gain, the IEEE 802.11a function Phy802_11LockAntennaDirection calls the generic antenna function ANTENNA_LockAntennaDirection. ANTENNA_LockAntennaDirection calls the antenna locking function for the antenna model that is in use. For example, if the patterned antenna is in use, ANTENNA_LockAntennaDirection calls function AntennaPatternedLockAntennaDirection. Phy802_11LockAntennaDirection is implemented in EXATA_HOME/libraries/wireless/src/phy_802_11.cpp. ANTENNA_LockAntennaDirection and the other generic antenna functions are implemented in antenna.cpp. AntennaPatternedLockAntennaDirection and the other functions for the patterned antenna are implemented in antenna_patterned.cpp.

Figure 4-180 shows the modifications that need to be made to ANTENNA_LockAntennaDirection to incorporate MYANTENNA in EXata. AntennaMyantennaLockAntennaDirection is the MYANTENNA function to lock the antenna direction.

```
void ANTENNA_LockAntennaDirection(Node* node, int phyIndex)
{
    PhyData* phyData = node->phyData[phyIndex];

    switch (phyData->antennaData->antennaModelType)
    {

        case ANTENNA_OMNIDIRECTIONAL:
        {
            break;
        }

        case ANTENNA_SWITCHED_BEAM:
        {
            AntennaSwitchedBeamLockAntennaDirection(node, phyIndex);
            break;
        }
        case ANTENNA_STEERABLE:
        {
            AntennaSteerableLockAntennaDirection(node, phyIndex);
            break;
        }

        case ANTENNA_PATTERNED:
        {
            AntennaPatternedLockAntennaDirection(node, phyIndex);
            break;
        }

    case ANTENNA_MYANTENNA:
        AntennaMyantennaLockAntennaDirection(node, phyIndex);
        break;

        default:
        {
            char err[MAX_STRING_LENGTH];
            sprintf(err, "Unknown ANTENNA-MODEL %s for phy %d.\n",
                phyData->antennaData->antennaModelType, phyIndex);
            ERROR_ReportError(err);
            break;
        }

    }//switch//
}
```

FIGURE 4-180. Modifying a Generic Antenna Function

Table 4-18 lists the generic antenna functions which should be modified to incorporate a new antenna model. Depending on the antenna's characteristics not all of the functions may need to be modified. The modifications to these functions are similar to the modification to ANTENNA_LockAntennaDirection, shown in Figure 4-180. These generic functions are implemented in antenna.cpp.

TABLE 4-18. Generic Antenna Functions

Function	Explanation
ANTENNA_IsInOmnidirectionalMode	Indicates if the antenna is operating in omni-directional mode.
ANTENNA_ReturnHeight	Returns the antenna height.
ANTENNA_ReturnPatternIndex	Returns the antenna pattern index.
ANTENNA_GainForThisDirection	Returns the antenna gain value for a specified direction.
ANTENNA_GainForThisDirectionWithPatternIndex	Returns the antenna gain value for a specified direction and pattern index.
ANTENNA_DefaultGainForThisSignal	Returns the default antenna gain value for a specified signal.
ANTENNA_LockAntennaDirection	For directional antennas, locks the direction in which maximum gain was observed.
ANTENNA_UnlockAntennaDirection	Unlocks the antenna's direction.
ANTENNA_IsLocked	Checks if the antenna's direction is locked.
ANTENNA_SetToDefaultMode	Sets the antenna to use the default mode.
ANTENNA_SetToBestGainConfigurationForThisSignal	If the antenna's direction is not locked, identifies the radiation pattern that provides the best gain for a given signal and uses it for further reception.
ANTENNA_SetBestConfigurationForAzimuth	Sets the antenna to the best configuration for a specified azimuth.

4.6.6.8 Implementing Antenna Functions

The functionality of an antenna model is implemented by means of several functions which are called by the PHY model and the propagation model. Write functions to implement the functionality of MYANTENNA. Include these functions in the antenna source file, antenna_myantenna.cpp, and define the prototypes of interface functions in the antenna header file, antenna_myantenna.h.

4.6.6.9 Integrating with PHY Models

Several Physical Layer and PHY model functions refer to antenna model functions and data structures directly. These functions should be modified to integrate MYANTENNA into EXata. Modifications required for these functions are shown in this section. Note that additional modifications to Physical Layer and PHY model functions may be necessary, depending upon the functionality of the antenna model and PHY models.

Figure 4-181 shows the modifications for function PHY_PropagationRange. It assumes that the MYANTENNA data structure, `AntennaMyantenna` (see Section 4.6.6.5), contains the fields `antennaGain_dB` and `antennaHeight`. PHY_PropagationRange is implemented in EXATA_HOME/libraries/wireless/src/phy.cpp.

```
double PHY_PropagationRange(Node* node,
                            int   interfaceIndex,
                            BOOL  printAllDataRates)
{
    ...
    AntennaOmnidirectional* omniDirectional;
    AntennaSwitchedBeam* switchedBeam;
    AntennaSteerable* steerable;
    AntennaPatterned* patterned;
    AntennaMyantenna* myantenna;
    ...
    switch (thisRadio->antennaData->antennaModelType)
    {
    case ANTENNA_OMNIDIRECTIONAL:
        {
            ...
        }
    ...
    case ANTENNA_PATTERNED:
        {
            patterned =
                (AntennaPatterned*)thisRadio->antennaData->antennaVar;
            txAntennaGain_dB = patterned->antennaGain_dB;
            txAntennaHeight = patterned->antennaHeight;
            break;
        }
    case ANTENNA_MYANTENNA:
        {
            myantenna =
                (AntennaMyantenna*)thisRadio->antennaData->antennaVar;
            txAntennaGain_dB = myantenna->antennaGain_dB;
            txAntennaHeight = myantenna->antennaHeight;
            break;
        }
    default:
        {
            ...
        }

    }
    ...
}
```

FIGURE 4-181. Modifications to Function PHY_PropagationRange

Figure 4-182 shows the modifications for function Phy802_11Init, which is implemented in phy_802_11.cpp.

```
void Phy802_11Init(
    Node *node,
    const int phyIndex,
    const NodeInput *nodeInput)
{
    BOOL   wasFound;
    BOOL   yes;
    int dataRateForBroadcast;
    int i;
    int numChannels = PROP_NumberChannels(node);
    ...
    // Antenna model initialization
    //
    ANTENNA_Init(node, phyIndex, nodeInput);

    ERROR_Assert(((phy802_11->thisPhy->antennaData->antennaModelType
                == ANTENNA_OMNIDIRECTIONAL) ||
           (phy802_11->thisPhy->antennaData->antennaModelType
                == ANTENNA_SWITCHED_BEAM) ||
           (phy802_11->thisPhy->antennaData->antennaModelType
                == ANTENNA_STEERABLE) ||
           (phy802_11->thisPhy->antennaData->antennaModelType
                == ANTENNA_MYANTENNA) ||
           (phy802_11->thisPhy->antennaData->antennaModelType
                == ANTENNA_PATTERNED)) ,
            "Illegal antennaModelType.\n");
    ...
    // Set PHY802_11-ESTIMATED-DIRECTIONAL-ANTENNA-GAIN
    //
    IO_ReadDouble(
        node->nodeId,
        node->phyData[phyIndex]->networkAddress,
        nodeInput,
        "PHY802.11-ESTIMATED-DIRECTIONAL-ANTENNA-GAIN",
        &wasFound,
        &(phy802_11->directionalAntennaGain_dB));

    if (!wasFound &&
        (phy802_11->thisPhy->antennaData->antennaModelType
            != ANTENNA_OMNIDIRECTIONAL &&
        (phy802_11->thisPhy->antennaData->antennaModelType
            != ANTENNA_MYANTENNA &&
          phy802_11->thisPhy->antennaData->antennaModelType
            != ANTENNA_PATTERNED))
    {
        ERROR_ReportError(
            "PHY802.11-ESTIMATED-DIRECTIONAL-ANTENNA-GAIN is missing\n");
    }
    ...
}
```

FIGURE 4-182. Modifications to Function Phy802_11Init

Figure 4-183 shows the modifications for function Phy802_11ITermintaeCurrentReceive, which is implemented in phy_802_11.cpp.

```
void Phy802_11TerminateCurrentReceive(
    Node* node, int phyIndex, const BOOL terminateOnlyOnReceiveError,
    BOOL* frameError,
    clocktype* endSignalTime)
{
    PhyData* thisPhy = node->phyData[phyIndex];
    PhyData802_11* phy802_11 = (PhyData802_11*)thisPhy->phyVar;
    ...
    *frameError = phy802_11->rxMsgError;

    if ((terminateOnlyOnReceiveError) && (!phy802_11->rxMsgError)) {
        return;
    }//if//
    if (thisPhy->antennaData->antennaModelType == ANTENNA_OMNIDIRECTIONAL) {
        phy802_11->interferencePower_mW += phy802_11->rxMsgPower_mW;
    }
    else {
        int channelIndex;
        PHY_GetTransmissionChannel(node, phyIndex, &channelIndex);
        ERROR_Assert(((thisPhy->antennaData->antennaModelType
                    == ANTENNA_SWITCHED_BEAM) ||
                (thisPhy->antennaData->antennaModelType
                    == ANTENNA_STEERABLE) ||
                (thisPhy->antennaData->antennaModelType
                    == ANTENNA_MYANTENNA) ||
                (thisPhy->antennaData->antennaModelType
                    == ANTENNA_PATTERNED)) ,
                 "Illegal antennaModelType");
        ...
        }
     ...
}
```

FIGURE 4-183. Modifications to Function Phy802_11TerminateCurrentReceive

Figure 4-184 shows the modifications for function Phy802_11IGetLastAngleOfArrival, which is implemented in phy_802_11.cpp. AntennaMyantennaGetLastBoresightAzimuth is the MYANTENNA function to return the last boresight azimuth angle.

```
double Phy802_11GetLastAngleOfArrival(Node* node, int phyIndex) {
    PhyData* thisPhy = node->phyData[phyIndex];

    switch (thisPhy->antennaData->antennaModelType) {

    case ANTENNA_SWITCHED_BEAM:
        {
            return AntennaSwitchedBeamGetLastBoresightAzimuth(node,
                phyIndex);
            break;
        }

    case ANTENNA_STEERABLE:
        {
            return AntennaSteerableGetLastBoresightAzimuth(node, phyIndex);
            break;
        }

    case ANTENNA_PATTERNED:
        {
            return AntennaPatternedGetLastBoresightAzimuth(node, phyIndex);
            break;
        }

    case ANTENNA_MYANTENNA:
        {
            return AntennaMyantennaGetLastBoresightAzimuth(node, phyIndex);
            break;
        }

    default:
        {
            ERROR_ReportError("AOA not supported for this Antenna Model\n");
            break;
        }
    }//switch//

    abort();
    return 0.0;
}
```

FIGURE 4-184. Modifications to Function Phy802_11GetLastAngleOfArrival

Figure 4-185 shows the modifications for function PhyAbstractInit, which is implemented in EXATA_HOME/libraries/wireless/src/phy_abstract.cpp.

```
void PhyAbstractInit(
    Node *node,
    const int phyIndex,
    const NodeInput *nodeInput)
{
    double rxSensitivity_dBm;
    double rxThreshold_dBm;
    ...
    //
    // Antenna model initialization
    //
    ANTENNA_Init(node, phyIndex, nodeInput);

    ERROR_Assert(((phy_abstract->thisPhy->antennaData->antennaModelType
                == ANTENNA_OMNIDIRECTIONAL) ||
           (phy_abstract->thisPhy->antennaData->antennaModelType
                == ANTENNA_MYANTENNA)) ||
           (phy_abstract->thisPhy->antennaData->antennaModelType
                == ANTENNA_PATTERNED)) ,
            "Illegal antennaModelType.\n");
   ...
}
```

FIGURE 4-185. Modifications to Function PhyAbstractInit

4.6.6.10 Including and Compiling Files

This step is similar to the one for adding a PHY model (see Section 4.6.5.12).

4.6.6.11 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.7 Communication Medium

The communication medium transmits signals between nodes. It interfaces with the Physical Layer entities at the nodes. A wireless communication medium model in EXata simulates the propagation of signals between nodes, taking into account both propagation delays and signal attenuation due to path loss, fading, and shadowing.

In EXata, a communication medium model has three components: a path loss model, a fading model, and a shadowing model. This section gives a detailed description of how to add each of these components to EXata.

4.7.1 Communication Medium Models in EXata

EXata provides several models for path loss, fading, shadowing, and channel interference.

Path Loss Models

Path loss refers to the attenuation of a signal in transit between a transmitter and receiver. Path loss may be due to many effects, such as free-space loss, refraction, reflection, aperture-medium coupling loss, and absorption. Table 4-19 lists the different pathloss models in EXata. See the corresponding model library for the description of each model and its parameters.

TABLE 4-19. Pathloss Models in EXata

Pathloss Model	Description	Model Library
COST231-HATA	COST 231-Hata pathloss model. This model can be used for urban, suburban, or open areas. It is a refinement of the Okumura-Hata pathloss model.	Urban Propagation
COST231-WALFISH-IKEGAMI	COST 231-Walfish-Ikegami pathloss model. This model can be used for urban or metropolitan areas.	Urban Propagation
FREE-SPACE	Friis free-space pathloss model. The model assumes an omni-directional line-of-sight propagation path. The signal strength decays with the square of the distance between the transmitter and receiver.	Wireless
ITM	Irregular Terrain Model, also known as the Longley-Rice model. This model uses the information from a terrain data file to calculate line-of-sight between nodes, ground reflection characteristics, and pathloss.	Wireless
OKUMURA-HATA	Okumura-Hata pathloss model for macro-cellular systems. This model can be used for urban, suburban, or open areas.	Urban Propagation
PATHLOSS-MATRIX	Matrix-based pathloss model. This model uses a four-dimensional matrix of pathloss values indexed by source node, destination node, simulation time, and channel number.	Wireless
STREET-M-TO-M	Street mobile-to-mobile pathloss model. This model calculates pathloss between a source and a destination in an urban canyon communicating across several building obstacles.	Urban Propagation

TABLE 4-19. Pathloss Models in EXata (Continued)

Pathloss Model	Description	Model Library
STREET-MICROCELL	Street micro-cell pathloss model. This model calculates the path-loss between transmitter-receiver pairs that are located in adjacent streets in an urban canyon.	Urban Propagation
SUBURBAN	Suburban pathloss model. This model characterizes propagation in a suburban environment and takes into account the effects of terrain and foliage on signals.	Urban Propagation
TIREM	Terrain Integrated Rough Earth Model. This model considers terrain effects, transmitter and receiver attributes such as antenna height and frequency, and atmospheric and ground constants. This model is distributed by the Joint Spectrum Center of the Department of Defense and is interfaced with EXata. This model requires a terrain data file.	TIREM Advanced Propagation
TWO-RAY	Two-ray pathloss model. The two-ray pathloss model considers a line-of-sight path and a reflection from flat earth in pathloss calculation.	Wireless

Fading Models

A fading model calculates the effect of changes in characteristics of the propagation path on the signal strength. Table 4-20 lists the different fading models in EXata. See the corresponding model library for the description of each model and its parameters.

TABLE 4-20. Fading Models in EXata

Fading Model	Description	Model Library
FAST-RAYLEIGH	Fast Rayleigh fading model. The fast Rayleigh fading model is a statistical model to represent the fast variation of signal amplitude at the receiver due to the motion of the transmitter/receiver pair.	Wireless
RAYLEIGH	Rayleigh fading model. Rayleigh fading model is a statistical model to represent the fast variation of signal amplitude at the receiver. In wireless propagation, Rayleigh fading occurs when there is no line of sight between the transmitter and receiver.	Wireless
RICEAN	Ricean fading model. This model can be used for scenarios where there is line of sight communication and the line of sight signal is the dominant signal seen at the receiver.	Wireless

Shadowing Models

A shadowing model calculates the attenuation caused to a signal by obstruction in the propagation path. Table 4-21 lists the different shadowing models in EXata. See the corresponding model library for the description of each model and its parameters.

TABLE 4-21. Shadowing Models in EXata

Shading Model	Description	Model Library
CONSTANT	Constant shadowing model. This model uses a constant shadowing offset.	Wireless
LOGNORMAL	Lognormal shadowing model. This model uses a lognormal distribution for the shadowing value.	Wireless

Inter-channel Interference Model

The Inter-channel Interference model accounts for both co-channel and inter-channel interference. (If the Inter-channel Interference model is not enabled only co-channel interference is taken into account.)

Co-channel interference occurs when the transmitting node and receiving nodes work on the same channel, the same channel index, the same frequency, and the same bandwidth. Inter-channel interference occurs when the transmitting node and receiving nodes work on different channels, different channel indexes, different frequencies, or different bandwidths, and there is frequency overlap between the channels. The Inter-channel Interference model estimates the effect of wide band jamming, narrow band jamming, frequency hopping jamming, frequency hopping spectrum spreading, frequency planning, spectrum analysis and spectrum management, etc.

See *Wireless Model Library* for details of the Inter-channel Interference model.

4.7.2 Communication Medium Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to communication medium models. These files contain detailed comments on functions and other code components.

Definitions of macros, functions, and structures relevant to communication medium models are contained in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/propagation.h

  This file contains definitions common to communication medium models and prototypes of functions defined in EXATA_HOME/libraries/wireless/src/propagation.cpp.
- EXATA_HOME/include/phy.h

  This file contains definitions of API functions needed to communicate with the Physical Layer.

Additionally, the following header file is also relevant to the communication medium:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.

The following are the folders and source files associated with the communication medium:

- EXATA_HOME/libraries/wireless/src

  This folder contains the source and header files for the various communication medium models implemented in EXata. The file names are based on the name of the model that they implement, e.g., to see the implementation for ITM path loss model, look at files prop_itm.cpp and prop_itm.h in this folder.

- EXATA_HOME/libraries/wireless/src/propagation.cpp

  This file contains the implementation of different communication medium models as well as generic communication medium functions.

4.7.3 Communication Medium Data Structures

The communication medium data structures are defined in EXATA_HOME/include/propagation.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file propagation.h for a complete description.)

1. `PathlossModel`: This is an enumeration type that lists all the path loss models.

```
enum PathlossModel {
    FREE_SPACE = 0,
    TWO_RAY,
    PL_MATRIX,
    ...
    FLAT_BINNING
};
```

2. `FadingModel`: This is an enumeration type that lists all the fading models.

```
enum FadingModel {
    NONE = 0,
    RICEAN
};
```

3. `ShadowingModel`: This is an enumeration type that lists all the shadowing models.

```
enum ShadowingModel {
    CONSTANT = 0,
    LOGNORMAL
};
```

4. `PropData`: This is the main data structure used by the communication medium and stores information about the propagation model used for each channel. Each node maintains an instance of this data structure for each channel. Some important fields of this structure are explained below.

```
struct PropData {
    int   numPhysListenable;
    int   numPhysListening;
    BOOL* phyListening;
    BOOL  limitedInterference;
    RandomDistribution<double> shadowingDistribution;
    int  nodeListId;
    int  numSignals;
    PropRxInfo* rxSignalList;
    double fadingStretchingFactor;
    PropPathProfile* pathProfile;
    void *propVar;
    int numPathLossCalculation;
};
```

- `numPhysListenable`: This is the number of wireless interfaces of the node that can potentially listen to this channel.
- `numPhysListening`: This is the number of wireless interfaces of the node that are currently listening to this channel.
- `phyListening`: This Boolean array indicates which of the node's wireless interfaces are currently listening to the channel.
- `shadowingDistribution`: This variable implements a random number distribution for use by the shadowing model.
- `nodeListId`: This is the list of nodes that can potentially listen to this channel.
- `numSignals`: This is the number of signals on this channel that the node is currently receiving, i.e., the number of other nodes that are currently transmitting on this channel and within whose propagation limit this node is located.
- `rxSignalList`: This list contains information on transmissions the node is currently receiving on this channel.
- `fadingStretchinFactor`: This variable determines the sampling interval used to read the fading trace.
- `pathProfile`: This data structure stores the characteristics of the path of the signal that the node is currently locked on to.

4.7.4 Communication Medium APIs and Communication with Physical Layer

This section describes the APIs used by the Physical Layer to communicate with the communication medium (see Section 4.7.4.1), the APIs used by the communication medium to communicate with the Physical Layer (see Section 4.7.4.2). This section also lists some of communication medium utility APIs (see Section 4.7.4.3).

The complete list of APIs, with their parameters and description, can be found in *API Reference Guide*.

4.7.4.1 Physical Layer to Communication Medium Communication

The communication medium provides the API PROP_ReleaseSignal to enable PHY entities to communicate with the communication medium. To transmit a signal, a PHY model calls the API PROP_ReleaseSignal.

The prototype for PROP_ReleaseSignal is contained in the file propagation.h.

4.7.4.2 Communication Medium to Physical Layer Communication

The communication medium uses the APIs listed below to communicate with PHY models. The prototypes for these functions are contained in EXATA_HOME/include/phy.h. The file EXATA_HOME/main/phy.cpp contains the implementation of these functions.

- PHY_SignalArrivalFromChannel: This function indicates the start of a signal.
- PHY_SignalEndFromChannel: This function indicates the end of a signal.

4.7.4.3 Communication Medium Utility APIs

Several APIs are available at the communication medium that perform tasks internal to the communication medium. Some of these functions can be used by other layers as well. The prototypes for these API functions are contained in the file propagation.h.

Some of the communication medium utility APIs are listed below.

- PROP_NumberChannels: This function return the number of channels.
- PROP_ChannelWavelength: This function returns the wavelength of the specified channel.

4.7.5 Adding a Path Loss Model

Although the working of each path loss model is different, there are certain functions that are performed by most path loss models. This section provides an outline for developing and adding a path loss model to EXata. We illustrate the process of adding a path loss model by using as an example the implementation code for the ITM path loss model. The header file for the ITM implementation is prop_itm.h and the source file is prop_itm.cpp in the folder EXATA_HOME/libraries/wireless/src. We use code snippets from these two files throughout this section to illustrate different steps in developing a path loss model. After understanding the discussed snippets, look at the complete code for ITM to understand how a path loss model is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a path loss model, MYPATHLOSS, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.7.5.2).
2. Modify the file propagation.cpp to include the model's header file (see Section 4.7.5.2).
3. Include the path loss model in the list of path loss models (see Section 4.7.5.3).
4. Decide on the format for the path loss model-specific configuration parameters (see Section 4.7.5.4.1).
5. Call the path loss model's initialization function from the propagation initialization function, PROP_GlobalInit (see Section 4.7.5.4.2).
6. Write the initialization function for the path loss model (see Section 4.7.5.4.3). The initialization function should read and store the configuration parameters for the path loss model.
7. Modify the propagation function PROP_CalculatePathloss to call the MYPATHLOSS's function to return the path loss value (see Section 4.7.5.5).
8. Implement the path loss calculation function for MYPATHLOSS (see Section 4.7.5.5).

9. Include the path loss model header and source files in the EXata tree and compile (see Section 4.7.5.6).

10. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.7.5.7).

4.7.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol or model, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a new path loss model, the programmer name the different components of the new model in a similar manner. It will be helpful to examine the implementation of the ITM model in EXata for hints for naming and coding different components of the new path loss model.

In this section, we describe the steps for developing a path loss model called “MYPATHLOSS”. We will use the string “Mypathloss” in the names of the different components of this model, just as the string “Itm” appears in the names of the components of the ITM implementation.

4.7.5.2 Creating Files

The first step towards adding a path loss model is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/wireless/src. However, it is recommended that all user-developed models be made part of an a library. In our example, we will place the path loss model in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

(If the model being developed is simple, then it may not be necessary to create separate files for it. In that case, code can be added directly to the files EXATA_HOME/include/propagation.h and EXATA_HOME/libraries/wireless/src/propagation.cpp. The rest of this section assumes that separate files for the path loss model will be created.)

If it doesn’t already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the path loss model and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *prop_* to designate the files as propagation model files.

Examples*:*

- prop_itm.h, prop_itm.cpp: These files, in the folder EXATA_HOME/libraries/wireless/src, implement the ITM path loss model.
- prop_plmatrix.h, prop_plmatrix.cpp: These files, in the folder EXATA_HOME/libraries/wireless/src, implement the path loss matrix model.

In keeping with the naming guidelines of Section 4.7.5.1, the header file for the example path loss model is called prop_mypathloss.h, and the source file is called prop_mypathloss.cpp.

It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, prop_mypathloss.h, should contain the following:

- Prototypes for interface functions in the source file, prop_mypathloss.cpp
- Constant definitions

The source file, prop_mypathloss.cpp, should contain the following:

- Statement to include the path loss model's header file:

```
#include "prop_mypathloss.h"
```

- Statements to include standard library functions and other header files needed by the path loss model's source file. A typical path loss model source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "api.h"
```

- Initialization function for the path loss model, MypathlossInitialize
- Path loss calculation function for the path loss model, PathlossMypathloss

The file EXATA_HOME/libraries/wireless/src/propagation.cpp contains the generic initialization function and function to calculate the path loss value. These generic functions in turn call the path loss model's initialization and path loss calculation functions. Therefore, to make these path loss model functions available to the generic functions, insert the following include statement in the file propagation.cpp, if separate files are created for the model:

```
#include "prop_mypathloss.h"
```

4.7.5.3 Including MYPATHLOSS in List of Path Loss Models

When a new path loss model is added to EXata, it needs to be included in the list of path loss models. To do this, add the path loss model's name to the enumeration `PathlossModel` defined in propagation.h (see Section 4.7.3).

For our example path loss model, add the entry `MYPATHLOSS` to `PathlossModel`, as shown in Figure 4-186.

```
typedef enum {
    FREE_SPACE = 0,
    TWO_RAY,
    PL_MATRIX,
    OPAR,
    ...
    FLAT_BINNING,
    MYPATHLOSS
} PathlossModel;
```

FIGURE 4-186. Adding MYPATHLOSS to List of Path Loss Models

Note **Always add to the end of lists in header files.**

4.7.5.4 Initialization

In this section, we describe the tasks that need to be performed as part of the initialization process of a path loss model.

4.7.5.4.1 Determining the Path Loss Model Configuration Format

A path loss model may use model-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a path loss model's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

As an example, the following parameters are used to specify the sampling distance and humidity for the ITM path loss model:

```
PROPAGATION-PATHLOSS-MODEL  ITM
PROPAGATION-SAMPLING-DISTANCE  100.0
PROPAGATION-HUMIDITY  10
```

Decide on the format for specifying the new path loss model's configuration parameters. For our example path loss model, specify the configuration parameters in the EXata configuration file using the following format (`<Identifier>` and `<Index>` can also be used to qualify the parameter declarations, as described above):

```
PROPAGATION-PATHLOSS-MODEL  MYPATHLOSS
<param1>                    <value1>
...
<paramN>                    <valueN>
```

where

`<param1>`, ..., `<paramN>`	: Names of parameters for `MYPATHLOSS`.
`<value1>`, ..., `<valueN`	: Values of the path loss parameters.

Section 4.7.5.4.3 explains how to read user input specified in this format to initialize the model.

4.7.5.4.2 Calling the Path Loss Model Initialization Function

The communication medium models are initialized before the protocol stack of each node is initialized. At the start of simulation, the initialization function PROP_GlobalInit is called, which calls the initialization function for the path loss models that are used in the simulation. PROP_GlobalInit is implemented in propagation.cpp.

To add your path loss model to EXata, make modifications to PROP_GlobalInit, as shown in Figure 4-187. MypathlossInitialize is the initialization function for MYPATHLOSS. Include the prototype of MypathlossInitialize in the header file, prop_mypathloss.h.

```
void PROP_GlobalInit(PartitionData *partitionData, NodeInput *nodeInput) {
    BOOL wasFound;
    char buf[MAX_STRING_LENGTH];
    PropChannel* propChannel;
    PropProfile* propProfile;
    ...
    for (i = 0; i < numChannels; i++) {
        ...
        //
        // Set pathlossModel
        //
        IO_ReadStringInstance(
            ANY_NODEID,
            ANY_ADDRESS,
            nodeInput,
            "PROPAGATION-PATHLOSS-MODEL",
            channelIndex,
            TRUE,
            &wasFound,
            buf);

        if (wasFound) {
            if (strcmp(buf, "FREE-SPACE") == 0) {
                propProfile->pathlossModel = FREE_SPACE;
            }
            ...
            else if (strcmp(buf, "ITM") == 0) {
                propProfile->pathlossModel = ITM;
                if (partitionData->terrainData.dataType == NO_TERRAIN_DATA) {
                    ERROR_ReportError("ITM requires terrain data\n");
                }
                ItmInitialize(
                    &(propChannel[channelIndex]), channelIndex, nodeInput);
            }
            else if (strcmp(buf, "MYPATHLOSS") == 0) {
                propProfile->pathlossModel = MYPATHLOSS;
                MypathlossInitialize(
                    &(propChannel[channelIndex]), channelIndex, nodeInput);
            }
            ...
        }
        else {
            ...
        }
        ...
    } //for//
    ...
}
```

FIGURE 4-187. Calling the Path Loss Model Initialization Function

4.7.5.4.3 Implementing the Path Loss Model Initialization Function

The initialization of a path loss model takes place in the initialization function of the model that is called by the function PROP_GlobalInit. The initialization function of a path loss model reads the user-specified parameters, if any, and sets the model parameters accordingly.

If your path loss model uses user-specified parameters, write an initialization function to read the values of these parameters. As an example, Figure 4-188 shows the initialization function for the ITM path loss model, ItmInitialize. ItmInitialize reads the values of user-specified parameters from the input file. If a parameter is not specified in the input file, ItmInitialize stores the default value for that parameter. ItmInitialize is implemented in EXATA_HOME/libraries/wireless/src/prop_itm.cpp.

The configurable parameters are read using IO functions such as IO_ReadDoubleInstance, IO_ReadIntInstance and IO_ReadStringInstance to read parameter values from the input file. IO_ReadDoubleInstance, IO_ReadIntInstance, IO_ReadStringInstance and other IO functions are defined in EXATA_HOME/include/fileio.h.

```
void ItmInitialize(
    PropChannel *propChannel,
    int channelIndex,
    const NodeInput *nodeInput)
{
    PropProfile* propProfile = propChannel->profile;
    BOOL wasFound;
    double elevationSamplingDistance;
    int climate;
    double refractivity;
    ...

    IO_ReadDoubleInstance(
        ANY_NODEID,
        ANY_ADDRESS,
        nodeInput,
        "PROPAGATION-SAMPLING-DISTANCE",
        channelIndex,
        (channelIndex == 0),
        &wasFound,
        &elevationSamplingDistance);

    if (wasFound) {
        propProfile->elevationSamplingDistance =
            (float)elevationSamplingDistance;
    }
    else {
        propProfile->elevationSamplingDistance =
            DEFAULT_SAMPLING_DISTANCE;
    }

    IO_ReadDoubleInstance(
        ANY_NODEID,
        ANY_ADDRESS,
        nodeInput,
        "PROPAGATION-REFRACTIVITY",
        channelIndex,
        (channelIndex == 0),
        &wasFound,
        &refractivity);

    if (wasFound) {
        propProfile->refractivity = refractivity;
    }
    else {
        propProfile->refractivity = DEFAULT_REFRACTIVITY;
    }
    ...
    return;
}
```

FIGURE 4-188. ITM Initialization Function

4.7.5.5 Path Loss Calculation

Function PROP_CalculatePathloss, implemented in propagation.cpp, is called by the communication medium to calculate the signal attenuation due to path loss between a pair of nodes.

To add a path loss model to EXata, modify function PROP_CalculatePathloss to call the model's path loss calculation function. Figure 4-189 shows the changes that need to be made to PROP_CalculatePathloss to add MYPATHLOSSMOEL to EXata. PathlossMypathloss is the MYPATHLOSS function that calculates the path loss. Include the prototype of PathlossMypathloss in the header file, prop_mypathloss.h.

Implement the path loss calculation function PathlossMypathloss and include it in the source file, prop_mypathloss.cpp.

```
void PROP_CalculatePathloss(
    Node* node,
    int channelIndex,
    double wavelength,
    float txAntennaHeight,
    float rxAntennaHeight,
    PropPathProfile *pathProfile,
    double* pathloss_dB)
{
    ...
    PropProfile *propProfile = node->propChannel[channelIndex].profile;
    ...
    switch (propProfile->pathlossModel) {
        case FREE_SPACE:
        case TWO_RAY:
        {
            ...
        }
        ...
        case ITM: {
            int numSamples;
            ...
            *pathloss_dB =
                        PathlossItm(
                            numSamples + 1,
                            pathProfile->distance / (double)numSamples,
                            elevationArray,
                            txPlatformHeight,
                            rxPlatformHeight,
                            propProfile->polarization,
                            propProfile->climate,
                            propProfile->permittivity,
                            propProfile->conductivity,
                            propProfile->frequency / 1.0e6,
                            propProfile->refractivity);
            ...
            return;
        }
        case MYPATHLOSS: {
            ...
            *pathloss_dB = PathlossMypathloss(...);
            return;
        }
        ...
        default: {
            abort();
        }
    }
    return;
}
```

FIGURE 4-189. Calling Path Loss Calculation Function

4.7.5.6 Including and Compiling Files

The final step in integrating your path loss model into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the path loss model in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Wireless library, then modify EXATA_HOME/libraries/wireless/Makefile-common as shown in Figure 4-190. Recompile EXata after making the changes.

```
...
# common sources
#
WIRELESS_SRCS = \
$(WIRELESS_DIR)/antenna.cpp \
$(WIRELESS_DIR)/antenna_global.cpp \
...
$(WIRELESS_DIR)/phy_802_11.cpp \
$(WIRELESS_DIR)/phy_abstract.cpp \
$(WIRELESS_DIR)/phy_cellular.cpp \
$(WIRELESS_DIR)/propagation.cpp \
$(WIRELESS_DIR)/prop_itm.cpp \
$(WIRELESS_DIR)/phy_mypathloss.cpp \
$(WIRELESS_DIR)/prop_plmatrix.cpp \
$(WIRELESS_DIR)/routing_aodv.cpp \
$(WIRELESS_DIR)/manet_packet.cpp \
...
```

FIGURE 4-190. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.7.5.7 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.7.6 Adding a Fading Model

This section describes how to add a fading model to EXata.

The following list summarizes the actions that need to be performed for adding a fading model, MYFADING, to EXata. Each of these steps is described in detail in subsequent sections.

1. Include the fading model in the list of fading models (see Section 4.7.6.1).
2. Decide on the format for the fading model-specific configuration parameters (see Section 4.7.6.2).
3. Modify function PROP_GlobalInit to include MYFADING and read its associated parameters, if any (see Section 4.7.6.3).
4. Modify function PROP_CalculateFading to calculate the fading value according to MYFADING (see Section 4.7.6.4).
5. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.7.6.5).

4.7.6.1 Including MYFADING in List of Fading Models

When a new fading model is added to EXata, it needs to be included in the list of fading models. To do this, add the fading model's name to the enumeration `FadingModel` defined in EXATA_HOME/include/propagation.h (see Section 4.7.3).

For our example fading model, add the entry `MYFADING` to `FadingModel`, as shown in Figure 4-191.

```
enum FadingModel {
    NONE = 0,
    RICEAN,
    MYFADING
};
```

FIGURE 4-191. Adding MYFADING to List of Fading Models

Note **Always add to the end of lists in header files.**

4.7.6.2 Determining the Fading Model Configuration Format

This step is similar to the one for adding a path loss model. See Section 4.7.5.4.1.

4.7.6.3 Initialization

The communication medium models are initialized before the protocol stack of each node is initialized. At the start of simulation, the initialization function PROP_GlobalInit is called, which reads the names of fading models and any associated parameters from the input file. PROP_GlobalInit is implemented in EXATA_HOME/libraries/wireless/src/propagation.cpp.

To add your fading model to EXata, make modifications to PROP_GlobalInit, as shown in Figure 4-192. Read any parameters used by MYFADING in PROP_GlobalInit. For example, if the Ricean fading model is specified, PROP_GlobalInit reads the K factor, which is specified as a configuration parameter, using the IO function IO_ReadDoubleInstance. IO_ReadDoubleInstance and other IO functions are defined in EXATA_HOME/include/fileio.h.

```
void PROP_GlobalInit(PartitionData *partitionData, NodeInput *nodeInput) {
    BOOL wasFound;
    char buf[MAX_STRING_LENGTH];
    PropChannel* propChannel;
    PropProfile* propProfile;
    ...
    for (i = 0; i < numChannels; i++) {
        ...
        //
        // Set fadingModel
        //
        IO_ReadStringInstance(
            ANY_NODEID, ANY_ADDRESS, nodeInput, "PROPAGATION-FADING-MODEL",
            channelIndex, TRUE, &wasFound, buf);
        if (wasFound) {
            if (strcmp(buf, "NONE") == 0) {
                propProfile->fadingModel = NONE;
            }
            else if (strcmp(buf, "RAYLEIGH") == 0) {
                ...
            }
            else if (strcmp(buf, "RICEAN") == 0) {
                propProfile->fadingModel = RICEAN;
                //
                // Set K factor
                //
                IO_ReadDoubleInstance(
                    ANY_NODEID, ANY_ADDRESS, nodeInput,
                    "PROPAGATION-RICEAN-K-FACTOR",
                    channelIndex, TRUE, &wasFound, &kFactor);
                if (wasFound) {
                    propProfile->kFactor = kFactor;
                }
                else {
                    ...
                }
            }
            else if (strcmp(buf, "FAST-RAYLEIGH") == 0) {
                ...
            }
            else if (strcmp(buf, "MYFADING") == 0) {
                propProfile->fadingModel = MYFADING;
                //
                // Read any configuration parameters used by MYFADING
            }
            ...
        }
        ...
    } //for//
    ...
}
```

FIGURE 4-192. Initializing Fading Models

4.7.6.4 Fading Calculation

Function PROP_CalculateFading, implemented in propagation.cpp, is called by the communication medium to calculate the signal attenuation due to fading between a pair of nodes.

To add a fading model to EXata, modify function PROP_CalculateFading to call the model's fading loss calculation function. Figure 4-193 shows the changes that need to be made to PROP_CalculateFading to add MYFADINGMOEL to EXata. FadingMyfading is the MYFADING function that calculates the fading loss.

Implement the fading calculation function FadingMyfading and include it in the source file, propagation.cpp.

```
void PROP_CalculateFading(
    Message* signalMsg,
    PropTxInfo* propTxInfo,
    Node* node2,
    int channelIndex,
    clocktype currentTime,
    float* fading_dB,
    double* channelReal,
    double* channelImag)
{
    PropChannel* propChannel = node2->partitionData->propChannel;
    PropProfile* propProfile = propChannel[channelIndex].profile;
    PropProfile* propProfile0 = propChannel[0].profile;

    if (propProfile->fadingModel == RICEAN) {
        int arrayIndex;
        double arrayIndexInDouble;
        double value1, value2;
        ...
    }
    else if (propProfile->fadingModel == MYFADING) {
        //
        // Calculating fading value.
        *fading_dB = FadingMyfading (...);
    }
    else {
        *fading_dB = 0.0;
    }
}
```

FIGURE 4-193. Fading Calculation Function

4.7.6.5 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.7.7 Adding a Shadowing Model

This section describes how to add a shadowing model to EXata.

In EXata, the shadowing loss is computed together with the path loss. The following list summarizes the actions that need to be performed for adding a shadowing model, MYSHADOWING, to EXata. Each of these steps is described in detail in subsequent sections.

1. Include the shadowing model in the list of shadowing models (see Section 4.7.7.1).
2. Modify function PROP_GlobalInit to include MYSHADOWING (see Section 4.7.7.2).
3. Modify function PROP_CalculatePathloss to calculate the shadowing loss according to MYSHADOWING (see Section 4.7.7.3).
4. To make the model available in the EXata GUI, modify the GUI settings files (see Section 4.7.7.4).

4.7.7.1 Including MYSHADOWING in List of Shadowing Models

When a new shadowing model is added to EXata, it needs to be included in the list of shadowing models. To do this, add the shadowing model's name to the enumeration `ShadowingModel` defined in EXATA_HOME/include/propagation.h (see Section 4.7.3).

For our example shadowing model, add the entry `MYSHADOWING` to `ShadowingModel`, as shown in Figure 4-194.

```
enum ShadowingModel {
    CONSTANT = 0,
    LOGNORMAL,
    MYSHADOWING
};
```

FIGURE 4-194. Adding MYSHADOWING to List of Shadowing Models

Note **Always add to the end of lists in header files.**

4.7.7.2 Initialization

The communication medium models are initialized before the protocol stack of each node is initialized. At the start of simulation, the initialization function PROP_GlobalInit is called, which reads the names of shadowing models from the input file. PROP_GlobalInit is implemented in EXATA_HOME/libraries/wireless/src/propagation.cpp.

To add your shadowing model to EXata, make modifications to PROP_GlobalInit, as shown in Figure 4-195.

```
void PROP_GlobalInit(PartitionData *partitionData, NodeInput *nodeInput) {
    BOOL wasFound;
    char buf[MAX_STRING_LENGTH];
    PropChannel* propChannel;
    PropProfile* propProfile;
    ...
    for (i = 0; i < numChannels; i++) {
        ...
        //
        // Set shadowingModel
        //
        IO_ReadStringInstance(
            ANY_NODEID, ANY_ADDRESS, nodeInput, "PROPAGATION-SHADOWING-MODEL",
            channelIndex, TRUE, &wasFound, buf);

        if (wasFound) {
            if (strcmp(buf, "NONE") == 0) {
                propProfile->shadowingModel = CONSTANT;
                propProfile->shadowingMean_dB = 0.0;
            }
            else {
                if (strcmp(buf, "LOGNORMAL") == 0) {
                    propProfile->shadowingModel = LOGNORMAL;
                }
                else if (strcmp(buf, "CONSTANT") == 0) {
                    propProfile->shadowingModel = CONSTANT;
                }
                else if (strcmp(buf, "MYSHADOWING") == 0) {
                    propProfile->shadowingModel = MYSHADOWING;
                }
                else {
                    char errorMessage[MAX_STRING_LENGTH];
                    sprintf(errorMessage,
                          "Error: unknown PROPAGATION-SHADOWING-MODEL '%s'.\n",
                            buf);
                    ERROR_ReportError(errorMessage);
                }
                ...
            }
        }
        else {
            propProfile->shadowingModel = CONSTANT;
            propProfile->shadowingMean_dB = PROP_DEFAULT_SHADOWING_MEAN_dB;
        }
        ...
    } //for//
    ...
}
```

FIGURE 4-195. Initializing Shadowing Models

4.7.7.3 Shadowing Loss Calculation

In EXata, the signal attenuation due to shadowing is calculated along with the path loss. This is done in function PROP_CalculatePathloss, implemented in propagation.cpp.

To add a shadowing model to EXata, modify function PROP_CalculatePathloss to call the model's shadowing loss calculation function. Figure 4-196 shows the changes that need to be made to PROP_CalculatePathloss to add MYSHADOWING to EXata. ShadowingMyshadowing is the MYSHADOWING function that calculates the shadowing loss.

Implement the shadowing loss calculation function ShadowingMyshadowing and include it in the source file, propagation.cpp.

```
void PROP_CalculatePathloss(
    Node* node,
    int channelIndex,
    double wavelength,
    float txAntennaHeight,
    float rxAntennaHeight,
    PropPathProfile *pathProfile,
    double* pathloss_dB)
{
    ...
    PropProfile *propProfile = node->propChannel[channelIndex].profile;
    ...
    switch (propProfile->pathlossModel) {
        case FREE_SPACE:
        case TWO_RAY:
        {
           double shadowing_dB = 0.0;
           if (propProfile->shadowingMean_dB != 0.0) {
                if (propProfile->shadowingModel == CONSTANT) {
                    shadowing_dB = propProfile->shadowingMean_dB;
                }
                else if (propProfile->shadowingModel == MYSHADOWING) {
                    //
                    // Calculate shadowing value
                    shadowing_dB = ShadowingMyshadowing(...);
                }
                else {
                    shadowing_dB =
                        propData->shadowingDistribution.getRandomNumber();
                }
            }
            if (propProfile->pathlossModel == FREE_SPACE) {
                *pathloss_dB = PROP_PathlossFreeSpace(pathProfile->distance,
                                                      wavelength);
            }
            else {
                assert(propProfile->pathlossModel == TWO_RAY);
                txPlatformHeight = pathProfile->fromPosition.common.c3 +
                                   txAntennaHeight;
                rxPlatformHeight = pathProfile->toPosition.common.c3 +
                                   rxAntennaHeight;
                *pathloss_dB = PROP_PathlossTwoRay(pathProfile->distance,
                                                   wavelength,
                                                   (float)txPlatformHeight,
                                                   (float)rxPlatformHeight);
            }
            *pathloss_dB += shadowing_dB;
            return;
        }
        ...
        }
    }
    return;
}
```

FIGURE 4-196. Calling Shadowing Loss Calculation Function

4.7.7.4 Integrating the Model into the GUI

To make the new model available in EXata GUI, modify the GUI settings files, as described in Section 5.1.4.

4.8 Node Mobility

In EXata, mobility models work together with node placement models and terrain models to simulate the mobility behavior of nodes. This section gives a detailed description of how to add a mobility model to EXata.

4.8.1 Mobility and Related Models in EXata

EXata provides several models for mobility, node placement, and terrain.

Mobility Models

A mobility model simulates the movement of a node or a group of nodes. Table 4-22 lists the different mobility models in EXata. See the corresponding model library for the description of each model and its parameters.

TABLE 4-22. Mobility Models in EXata

Mobility Model	Description	Model Library
FILE	File-based mobility model The node positions at different simulation times are read from a file. The node moves from one position to the next in a straight line at a constant speed.	Wireless
GROUP	Group-based mobility model. In this model, groups of nodes move together. The entire group moves following the Random Waypoint model, and each node moves within the group area, also following the Random Waypoint model.	Wireless
RANDOM-WAYPOINT	Random Waypoint mobility model. The node selects a random position, moves towards it in a straight line at a constant speed that is randomly selected from a range, and pauses at that destination. The node repeats this process. throughout the simulation.	Wireless

Node Placement Models

A node placement model determines the initial positions of nodes in a simulation. Table 4-23 describes the different node-placement models in EXata. See the corresponding model library for the description of each model and its parameters.

TABLE 4-23. Node Placement Models in EXata

Node Placement Model	Description	Model Library
FILE	File-based node placement policy. The initial node positions are read from a file.	Developer
GRID	Grid node placement policy. The terrain is divided into a number of squares. One node is placed at each grid point. **Note**: This model can be used only if the number of nodes in the scenario is a square of an integer (4, 9, 16, ...).	Developer

TABLE 4-23. Node Placement Models in EXata (Continued)

Node Placement Model	Description	Model Library
GROUP	Group-based node placement policy. This node placement model is used with the group mobility model.	Wireless
RANDOM	Random node placement policy. Nodes are placed on the terrain randomly.	Developer
UNIFORM	Uniform node placement policy. The terrain is divided into a number of equal-sized square cells. One node is placed in each cell randomly.	Developer

In EXata mobility models, the Cartesian coordinate system is used for small areas where the curvature of the earth can be ignored, and the spherical coordinate system (latitude-longitude-altitude) is used for larger areas terrain where the curvature of the earth cannot be ignored.

The mobility behavior is defined by the following rules:

- The mobility model (except for the pedestrian mobility model) specifies an array of destinations and arrival times for each node.
- Each node moves from its current position towards the next destination along a straight line. EXata determines the intermediate positions at user specified distances.
- Elevation of each node with ground mobility models is determined by the terrain data, if available and if requested by the user.
- Mobility includes node orientation (both horizontal and vertical).

4.8.2 Mobility Models Organization: Files and Folders

In this section, we briefly examine the files and folders that are relevant to mobility models. These files contain detailed comments on functions and other code components.

Definitions of macros, functions, and structures relevant to mobility models are contained in the following header files:

- EXATA_HOME/include/api.h

  This file defines the events and data structures needed to communicate between different layers of the protocol stack.
- EXATA_HOME/include/mobility.h

  This file contains definitions common to mobility models and prototypes of functions defined in EXATA_HOME/main/mobility.cpp.

Additionally, the following header file is also relevant to mobility models:

- EXATA_HOME/include/fileio.h

  This file contains prototypes of functions to read input files and create output files.

The following are the folders and source files associated with mobility models:

- EXATA_HOME/libraries/wireless/src and EXATA_HOME/libraries/developer/src

  These folders contain the source and header files for the various mobility models implemented in EXata. The file names are based on the name of the model that they implement, e.g., to see the

implementation for the random waypoint mobility model, look at files mobility_waypoint.cpp and mobility_waypoint.h in the folder EXATA_HOME/libraries/wireless/src.

- EXATA_HOME/main/mobility.cpp

  This file contains the implementation of the generic mobility functions.

- EXATA_HOME/libraries/developer/src/mobility_placement.cpp

  This file contains the implementation of the generic node placement functions.

4.8.3 Mobility-related Data Structures

The mobility-related data structures are defined in EXATA_HOME/include/mobility.h. This section describes the main data structures. (Note that only a partial description of the data structures is provided here. Refer to file mobility.h for a complete description.)

1. `MobilityType`: This is an enumeration type that lists all the mobility models.

```
typedef enum {
    NO_MOBILITY = 0,
    RANDOM_WAYPOINT_MOBILITY,
    FILE_BASED_MOBILITY,
    GROUP_MOBILITY,
    PEDESTRIAN_MOBILITY
} MobilityType;
```

2. `MobilityElement`: This data structure stores the coordinates and time of arrival at a point in a node's path.

```
struct MobilityElement {
    int               sequenceNum;
    clocktype         time;
    Coordinates       position;
    Orientation       orientation;
    double            speed;
};
```

3. `MobilityRemainder`: This data structure stores information needed to compute the coordinates and times of arrival at intermediate points in a node's path.

```
struct MobilityRemainder {
    clocktype   nextMoveTime;
    Coordinates nextPosition;
    Orientation nextOrientation;
    double      speed;
    int         numMovesToNextDest;
    int         destCounter;
    clocktype   moveInterval;
    Coordinates delta;
};
```

4. `MobilityData`: This is the main data structure that stores mobility information for a node. Some important fields of this structure are explained below.

```
struct MobilityData {
    MobilityType      mobilityType;
    D_Float32         distanceGranularity;
    D_BOOL            groundNode;
    BOOL              mobilityStats;
    RandomSeed        seed;
    int               sequenceNum;
    MobilityElement*  next;
    MobilityElement*  current;
    MobilityElement*  past[NUM_PAST_MOBILITY_EVENTS];
    int               numDests;
    MobilityElement*  destArray;
    MobilityRemainder remainder;
    clocktype         lastExternalTrueMobilityTime;
    clocktype         lastExternalMobilityTime;
    Velocity          lastExternalVelocity;
    double            lastExternalSpeed;
    bool              indoors;
    PedestrianData*   pedestrianData;
    void              *mobilityVar;
};
```

- `mobilityType`: This indicates the mobility model used by the node.
- `distanceGranularity`: This variable determines how frequently a node's position is updated by the simulator.
- `groundNode`: This variable indicates whether the node is on the ground.
- `mobilityStats`: This flag indicates whether mobility statistics collection is enabled.
- `seed`: The variable stores the seed to be used by the mobility model.
- `sequenceNum`: This variable stores the sequence number of mobility events.
- `next`, `current`: These variables store the next and current positions of the node, respectively.
- `past`: This an array of the most recent `NUM_PAST_MOBILITY_EVENTS` positions of the node.
- `numDests`: This variable stores the number of destinations.
- `destArray`: This array stores the coordinates and time of arrival for the node positions as calculated by the mobility model.
- `remainder`: This structure stores information to determine the coordinates and time of arrival at intermediate points along the current segment of the node's trajectory.
- `lastExternalTrueMobilityTime`: This variable stores the last external true mobility time.
- `lastExternalMobilityTime`: This variable stores the last external mobility time.
- `last External Velocity`: This variable stores the last external moving velocity.
- `lastExternalSpeed`: This variable stores the last external moving speed.
- `pedestrianData`: The is a pointer to the data structure for the pedestrian mobility model.
- `mobilityVar`: The is a pointer to the data structure for the mobility variable.

4.8.4 Mobility APIs

Several API functions are available for different mobility models to perform common mobility-related tasks. The prototypes for these functions are contained in the file mobility.h. The file EXATA_HOME/main/mobility.cpp contains the implementation of these functions.

The complete list of APIs, with their parameters and description, can also be found in *API Reference Guide*. Some of the mobility-related APIs are listed below.

- MOBILITY_AddANewDestination: This function adds a new destination for a node.
- MOBILITY_NextMoveTime: This function returns the time of the next move by a node.
- MOBILITY_ReturnCoordinates: This function returns the coordinates of a node.

4.8.5 Adding a Mobility Model

Although the working of each mobility model is different, there are certain functions that are performed by most mobility models. This section provides an outline for developing and adding a mobility model to EXata. We illustrate the process of adding a mobility model by using as an example the implementation code for the random waypoint mobility model. The implementation files for the random waypoint are mobility_waypoint.h and mobility_waypoint.cpp in the folder EXATA_HOME/libraries/wireless/src. We use code snippets from these two files throughout this section to illustrate different steps in developing a mobility model. After understanding the discussed snippets, look at the complete code for the random waypoint mobility model to understand how a mobility model is implemented in EXata.

The following list summarizes the actions that need to be performed for adding a mobility model, MYMOBILITY, to EXata. Each of these steps is described in detail in subsequent sections.

1. Create header and source files (see Section 4.8.5.2).
2. Modify the files mobility.cpp and mobility_placement.cpp to include the model's header file (see Section 4.8.5.2).
3. Include the mobility model in the list of mobility models (see Section 4.8.5.3).
4. Decide on the format for the mobility model-specific configuration parameters (see Section 4.8.5.4).
5. Modify the mobility initialization function MOBILITY_PreInitialize to read the mobility model's name from the input file (see Section 4.8.5.5).
6. Modify the generic mobility function SetRandomMobility to call MYMOBILITY's position calculation function (see Section 4.8.5.5).
7. Implement the mobility model function (see Section 4.8.5.6). In general, the mobility model function should perform the following tasks:
   a. Read and store the user-specified configuration parameters for the mobility model.
   b. Calculate and store the node positions using the mobility model's algorithm.
8. Include the mobility model header and source files in the EXata tree and compile (see Section 4.8.5.7).

4.8.5.1 Naming Guidelines

In EXata, each component (file, data structure, function, etc.) is given a name that indicates the name of the protocol or model, the layer in which the protocol resides, and the functionality of the component, as appropriate. We recommend that when adding a mobility model, the programmer name the different components of the new model in a similar manner. It will be helpful to examine the implementation of the random waypoint model in EXata for hints for naming and coding different components of the new mobility model.

In this section, we describe the steps for developing a mobility model called “MYMOBILITY”. We will use the string “Mymobility” in the names of the different components of this model, just as the string “Waypoint” appears in the names of the components of the random waypoint implementation.

4.8.5.2 Creating Files

The first step towards adding a mobility model is creating files. Most models comprise two files: the header file and the source file. These files can be placed in any library, e.g., in the folder EXATA_HOME/libraries/wireless/src. However, it is recommended that all user-developed models be made part of a library. In our example, we will place the mobility model in a library called user_models. See Section 4.10 for instructions for creating and activating a library.

If it doesn’t already exist, create a directory in EXATA_HOME/libraries called user_models and a subdirectory in EXATA_HOME/libraries/user_models called src. Create the files for the mobility model and place them in the folder EXATA_HOME/libraries/user_models/src. Name these files in a way that clearly indicates the model that they implement. Prefix the file names with *mobility_* to designate the files as mobility model files.

Examples*:*

- mobility_waypoint.h, mobility_waypoint.cpp: Implement the random waypoint mobility model
- mobility_group.h, mobility_group.cpp: Implement the group mobility model

In keeping with the naming guidelines of Section 4.8.5.1, the header file for the example mobility model is called mobility_mymobility.h, and the source file is called mobility_mymobility.cpp.

Note **It is strongly recommended to have separate header and source files. Not having a header file may lead to unexpected problems even if the compilation process does not indicate any error.**

While adding code to the files, it is important to organize the code well between the files. Generally, the header file, mobility_mymobility.h, should contain the following:

- Prototypes for interface functions in the source file, mobility_mymobility.cpp
- Constant definitions

The source file, mobility_mymobility.cpp, should contain the following:

- Statement to include the mobility model’s header file:

```
#include “mobility_mymobility.h”
```

- Statements to include standard library functions and other header files needed by the mobility model’s source file. A typical mobility model source file includes the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "api.h"
#include "partition.h"
```

- Function to calculate node positions over time, MOBILITY_MymobilityInit

The files EXATA_HOME/main/mobility.cpp and EXATA_HOME/libraries/developer/src/mobility_placement.cpp contain the generic functions that in turn call the specific mobility models’

functions. Therefore, to make the mobility model functions available to the generic functions, insert the following include statement in the files mobility.cpp and mobility_placement.cpp:

```
#include "mobility_mymobility.h"
```

4.8.5.3 Including MYMOBILITY in List of Mobility Models

When a mobility model is added to EXata, it needs to be included in the list of mobility models. To do this, add the mobility model's name to the enumeration `MobilityType` defined in mobility.h (see Section 4.8.3).

For our example mobility model, add the entry `MYMOBILITY` to `MobilityType`, as shown in Figure 4-197.

```
typedef enum {
    NO_MOBILITY = 0,
    RANDOM_WAYPOINT_MOBILITY,
    FILE_BASED_MOBILITY,
    GROUP_MOBILITY,
    PEDESTRIAN_MOBILITY,
    MYMOBILITY
} MobilityType;
```

FIGURE 4-197. Adding MYMOBILITY to List of Mobility Models

Note **Always add to the end of lists in header files.**

4.8.5.4 Determining the Mobility Model Configuration Format

A mobility model may use model-specific configuration parameters. The configuration parameters are specified in the EXata configuration file. The format for specifying a mobility model's configuration parameters is:

```
[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>
```

where

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

As an example, the following parameters specify the pause time, minimum speed and maximum speed for the random waypoint mobility model:

```
MOBILITY                RANDOM_WAYPOINT
MOBILITY-WP-PAUSE       30S
MOBILITY-WP-MIN-SPEED   0
MOBILITY-WP-MAX-SPEED   10
```

Decide on the format for specifying the new mobility model's configuration parameters. For our example mobility model, specify the configuration parameters in the EXata configuration file using the following format (<Identifier> and <Index> can also be used to qualify the parameter declarations, as described above):

```
MOBILITY     MYMOBILITY
<param1>     <value1>
...
<paramN>     <valueN>
```

where

<param1>, ..., <paramN> : Names of parameters for MYMOBILITY.

<value1>, ..., <valueN : Values of the mobility parameters.

Section 4.8.5.6 explains how to read user input specified in this format to initialize the model.

4.8.5.5 Modifying Generic Mobility Functions

To add a new mobility model to EXata, two generic mobility functions need to be modified: MOBILITY_PreInitialize and SetRandomMobility.

Function MOBILITY_PreInitialize, implemented in mobility.cpp, is called for each node at the start of simulation. MOBILITY_PreInitialize reads the name of the mobility model specified for a node in the configuration file, and stores it in the mobilityType field of the mobilityData data structure associated with the node (see Section 4.8.3). To add MYMOBILITY to EXata, modify MOBILITY_PreInitialize as shown in Figure 4-198.

```
void MOBILITY_PreInitialize(
    NodeAddress nodeId,
    MobilityData* mobilityData,
    NodeInput* nodeInput,
    int seedVal)
{
    int i;
    BOOL wasFound;
    char buf[MAX_STRING_LENGTH];
    // Set mobilityType.
    IO_ReadString(
        nodeId,
        ANY_ADDRESS,
        nodeInput,
        "MOBILITY",
        &wasFound,
        buf);
    if (wasFound) {
        if (strcmp(buf, "NONE") == 0) {
            mobilityData->mobilityType = NO_MOBILITY;
        }
        else if (strcmp(buf, "RANDOM-WAYPOINT") == 0) {
            mobilityData->mobilityType = RANDOM_WAYPOINT_MOBILITY;
        }
        else if (strcmp(buf, "GROUP-MOBILITY") == 0) {
            mobilityData->mobilityType = GROUP_MOBILITY;
        }
        else if (strcmp(buf, "MYMOBILITY") == 0) {
            mobilityData->mobilityType = MYMOBILITY;
        }
        ...
        else if (strcmp(buf, "FILE") == 0) {
            mobilityData->mobilityType = FILE_BASED_MOBILITY;
        }
        else {
            char errorMessage[MAX_STRING_LENGTH];
            sprintf(errorMessage, "Unknown MOBILITY type: %s.\n", buf);
            ERROR_ReportError(errorMessage);
        }
    }
    else {
        mobilityData->mobilityType = NO_MOBILITY;
    }
    ...
}
```

FIGURE 4-198. Reading Mobility Model's Name from Input File

Function SetRandomMobility, implemented in mobility_placement.cpp, is called for each node at the start of simulation to calculate the node positions over time. These positions, or destinations, are stored in the array, `destArray` in the node's mobility data structure, `MobilityData`. Between any two consecutive destinations, the node moves along a straight line. SetRandomMobility, in turn calls the position calculation function for the mobility model specified for the node. For example, if random waypoint mobility model is specified for a node, SetRandomMobility calls function MOBILITY_WaypointInit. Mobility_WaypointInit is implemented in mobility_waypoint.cpp. To add MYMOBILITY to EXata, call MYMOBILITY's node position calculation function, MOBILITY_MymobilityInit, from SetRandomMobility, as shown in Figure 4-199.

```
static
void SetRandomMobility(
    NodeAddress nodeId,
    MobilityData* mobilityData,
    TerrainData* terrainData,
    NodeInput* nodeInput,
    clocktype maxSimTime)
{
    assert(mobilityData->mobilityType != FILE_BASED_MOBILITY);

    if (mobilityData->mobilityType == NO_MOBILITY) {
        if (mobilityData->numDests > 1) {
            mobilityData->numDests = 1;
        }
    }
    else if (mobilityData->mobilityType == RANDOM_WAYPOINT_MOBILITY) {
        MOBILITY_WaypointInit(
            nodeId, mobilityData, terrainData, nodeInput, maxSimTime);
    }
    else if (mobilityData->mobilityType == MYMOBILITY) {
        MOBILITY_MymobilityInit(
            nodeId, mobilityData, terrainData, nodeInput, maxSimTime);
    }
}
```

FIGURE 4-199. Calling Mobility Model Position Calculation Function

4.8.5.6 Implementing Mobility Model Functions

A mobility model reads the mobility-related configuration parameters from the input file. It then calculates the position of the node at different times, and stores the positions and associated times of arrival in an array.

Figure 4-200 and Figure 4-201 show the implementation of the random waypoint mobility function, MOBILITY_WaypointInit, which is implemented in EXATA_HOME/libraries/wireless/src/mobility_waypoint.cpp. MOBILITY_WaypointInit first reads the mobility parameters, `MOBILITY-WP-PAUSE`, `MOBILITY-WP-MIN`, and `MOBILITY-WP-MAX`, and then calculates the future positions of the node according to the specified algorithm. The configurable parameters are read using IO functions such as IO_ReadString and IO_ReadDouble to read parameter values from the input file. IO_ReadString, IO_ReadDouble and other IO functions are defined in EXATA_HOME/include/fileio.h.

```
void MOBILITY_WaypointInit(
    NodeAddress nodeId,
    MobilityData* mobilityData,
    TerrainData* terrainData,
    NodeInput *nodeInput,
    clocktype maxSimClock)
{
    clocktype        simClock;
    Coordinates dest1;
    Coordinates dest2;
    Orientation orientation;
    clocktype mobilityPause;
    double    minSpeed;
    double    maxSpeed;
    char      buf[MAX_STRING_LENGTH];
    BOOL      wasFound;
    // no orientation for this model right now.
    orientation.azimuth = 0;
    orientation.elevation = 0;
    /* Read the pause time after reaching destination */
    IO_ReadString(
        nodeId,
        ANY_ADDRESS,
        nodeInput,
        "MOBILITY-WP-PAUSE",
        &wasFound,
        buf);
    assert(wasFound == TRUE);
    mobilityPause = TIME_ConvertToClock(buf);
    /* Read the speed arrange(Min,Max) */
    IO_ReadDouble(
        nodeId,
        ANY_ADDRESS,
        nodeInput,
        "MOBILITY-WP-MIN-SPEED",
        &wasFound,
        &minSpeed);
    assert(wasFound == TRUE);
    IO_ReadDouble(
        nodeId,
        ANY_ADDRESS,
        nodeInput,
        "MOBILITY-WP-MAX-SPEED",
        &wasFound,
        &maxSpeed);
    assert(wasFound == TRUE);
    ...
}
```

FIGURE 4-200. Random Waypoint Mobility Function: Reading Mobility Parameters

```
void MOBILITY_WaypointInit(
    NodeAddress nodeId,
    MobilityData* mobilityData,
    TerrainData* terrainData,
    NodeInput *nodeInput,
    clocktype maxSimClock)
{
    ...
    simClock = 0;
    dest1 = mobilityData->current->position;

    while (simClock < maxSimClock) {
        double           distance;
        double           speed;

        dest2.common.c1 =
            terrainData->origin.common.c1 +
            (terrainData->dimensions.common.c1 *
                                    RANDOM_erand(mobilityData->seed));
        dest2.common.c2 =
            terrainData->origin.common.c2 +
            (terrainData->dimensions.common.c2 *
                                   RANDOM_erand(mobilityData->seed));
        dest2.common.c3 = 0.0;
        // This model assumes that the third coordinate is always 0.0
        COORD_CalcDistance(
            terrainData->coordinateSystemType,
            &dest1, &dest2, &distance);
        speed = minSpeed
                + (RANDOM_erand(mobilityData->seed) * (maxSpeed - minSpeed));
        simClock += (clocktype)(distance / speed * SECOND);
        if (mobilityData->groundNode == TRUE) {
            MOBILITY_GetGroundElevation(terrainData, &dest2);
        }
        MOBILITY_AddANewDestination(
                               mobilityData, simClock, dest2, orientation);
        simClock += mobilityPause;
        if (mobilityPause > 0) {
            MOBILITY_AddANewDestination(
                mobilityData, simClock, dest2, orientation);
        }
        dest1 = dest2;
    }
    return;
}
```

FIGURE 4-201. Random Waypoint Mobility Function: Calculating Node Positions

4.8.5.7 Including and Compiling Files

The final step in integrating your mobility model into EXata is to add the source file to the EXata source tree and compile.

If you have created the files for the mobility model in an existing library or addon, then add the source file to the Makefile-common for that library or addon. For example, if you have created your model files in the Wireless library, then modify EXATA_HOME/libraries/wireless/Makefile-common as shown in Figure 4-202. Recompile EXata after making the changes.

```
...
# common sources
#
WIRELESS_SRCS = \
$(WIRELESS_DIR)/antenna.cpp \
$(WIRELESS_DIR)/antenna_global.cpp \
...
$(WIRELESS_DIR)/mac_maca.cpp \
$(WIRELESS_DIR)/mac_tdma.cpp \
$(WIRELESS_DIR)/mobility_group.cpp \
$(WIRELESS_DIR)/mobility_mymobility.cpp \
$(WIRELESS_DIR)/mobility_pedestrian.cpp \
$(WIRELESS_DIR)/mobility_waypoint.cpp \
$(WIRELESS_DIR)/multicast_odmrp.cpp \
...
```

FIGURE 4-202. Adding Model to Makefile-common

If you have created a new library called user_models, then follow the instructions given in Section 4.10.5 to integrate the user_models library into EXata.

4.9 Adding Trace Collection

EXata provides tracing capabilities which enable a user to trace a packet as it traverses the protocol stack at each node in the network that the packet visits. The packet trace lists information such as the node identifier, the layer in the stack and the protocol, among other things. This section describes how to add tracing capability for a custom-built protocol.

The following list summarizes the actions that need to be performed to add tracing capability to a user-developed protocol, MYPROTOCOL. Each of these steps is described in detail in subsequent sections.

1. Add the protocol to the list of traceable protocols (see Section 4.9.2).
2. In the initialization function of the protocol enable or disable tracing for the protocol, as specified in the configuration file (see Section 4.9.3).
3. Write a function to print the new protocol's header (see Section 4.9.4).
4. Make calls to the trace function TRACE_PrintTrace at appropriate places in the code to trace a packet (see Section 4.9.5).

4.9.1 Trace File Format

This section describes the format of the trace file produced by Simulator. The trace file contains a header which describes parameters for the experiment, a protocol map which gives a mapping between the protocols being traced and their integer identifiers, and one or more records, where each record corresponds to a single packet trace.

The format of a trace file is informally described below. Figure 4-203 shows an example of a trace file generated by Simulator. The XML definition file shown in Figure 4-204 describes the format of a trace file more formally.

```
<trace-file>
<head>
<version>  Kernel_Version </version>
<scenario> Scenario_Name </scenario>
<comments> Comments </comments>
</head>
<body>
Protocol_Map
Records
</body>
</trace_file>
```

The different elements used in the above format definition are described below.

- *Kernel_Version*: String indicating the kernel version of EXata, e.g., "`12.10`"
- *Scenario_Name*: String indicating the name of the scenario, e.g., "`trace-tcp`"
- *Comments*: String containing any user or system comments

- *Protocol_Map*: One or more occurrences of

  `<protocol_map>` *Protocol_Id Protocol_Name* `</protocol_map>`

  where

  Protocol_Id : Integer identifier used for the protocol *Protocol_Name* in the trace. This is the same as the protocol's integer value in the enumeration `TraceProtocolType` in EXATA_HOME/include/trace.h.

  Protocol_Name : Name of the protocol being traced

  The following are examples of *Protocol_Map*:

  ```
  <protocol_map>3 IPv4</protocol_map>
  <protocol_map>1 TCP</protocol_map>
  ```

- *Records*: One or more occurrences of

  ```
  <rec>
  <rechdr> Record_Header </rechdr>
  <recbody>
  Record_Body
  </recbody>
  </rec>
  ```

  Record_Header and *Record_Body* are defined below.

- *Record_Header*: Record header, which has the following format

  Originating_Node_Id Message_Seq_No Simulation_Time Originating_Protocol_Id Processing_Node_Id Tracing_Protocol_Id Action_Description

  where

  Originating_Node_Id : Identifier of the node where the packet originated

  Message_Seq_No : Sequence number of the message

  Simulation_Time : Current simulation time in seconds

  Originating_Protocol_Id : Identifier of the protocol that created the packet, i.e., the integer value of the protocol in the enumeration `TraceProtocolType` in trace.h

  Processing_Node_Id : Identifier of the current node, i.e., the node printing this record

  Tracing_Protocol_Id : Identifier of the protocol that is generating this trace record

  Action_Description : Description of the packet action that triggered the trace. This is described in detail below.

  The following are examples of *Record_Header*:

  ```
  3 0 0.033584881 7 3 2 <action> 1 0</action>
  3 0 0.033584881 7 3 3 <action> 4 <queue> 0 192</queue></action>
  ```

- *Action_Description*: Description of the action that triggered the trace. It has the following format:

  `<action>` *Action* `</action>`

  where *Action* is one of the following:
  - *Send_Action Comment*
  - *Receive_Action Comment*
  - *Drop_Action Comment*
  - *Enqueue_Action <queue> Queue_Interface_Id Queue_Priority </queue>*
  - *Dequeue_Action* `<queue>` *Queue_Interface_Id Queue_Priority* `</queue>`

  where

Send_Action	: Integer code for a packet send (“1”)
Receive_Action	: Integer code for a packet receive (“2”)
Drop_Action	: Integer code for a packet drop (“3”)
Enqueue_Action	: Integer code for a packet enqueue (“4”)
Dequeue_Action	: Integer code for a packet dequeue (“5”)
Comment	: Integer code for the comment explaining the send, receive or drop action. The possible comments are listed in the enumeration `PacketActionCommentType` in trace.h.
Queue_Interface_Id	: Interface index for the queue
Queue_Priority	: Queue priority

  The following are examples of *Action_Description*:

  ```
  <action> 2 0</action>
  <action> 5 <queue> 0 2</queue></action>
  ```

- *Record_Body*: One or more occurrences of

  Header_Start_Delimiter Header_Fields Header_End_Delimiter

  where

Header_Start_Delimiter	: String indicating the start of a header, which is the protocol name enclosed between “<“ and “>”. Examples are “`<udp>`” and “`<ipv4>`”.
Header_End_Delimiter	: String indicating the end of a header, which is the protocol name enclosed between “`</`“ and “>”. Examples are “`</udp>`” and “`</ipv4>`”.
Header_Fields	: List of the values of the header fields. The fields are separated by spaces. The list of flags in a header is enclosed between the delimiters “`<flags>`” and “`</flags>`”.

  The following are examples of *Record_Body*:

  ```
  <udp>519 519 36 0</udp>
  <ipv4>4 5 48 0 0 56 0 <flags>0 0 0</flags> 0 64 17 0 255.255.255.255
  192.0.1.255</ipv4>
  ```

Figure 4-203 shows an example trace file generated by Simulator.

```
<trace_file>

<head>
<version>12.10</version>
<scenario>trace-tcp</scenario>
<comments>Any user or system free-form comments</comments>
</head>

<body>

<protocol_map>3 IPv4</protocol_map>
<protocol_map>1 TCP</protocol_map>
<protocol_map>2 UDP</protocol_map>

<rec>
<rechdr> 3 0 0.033584881 7 3 2 <action> 1 0</action></rechdr>
<recbody>
<udp>519 519 36 0</udp>
</recbody>
</rec>

<rec>
<rechdr> 3 0 0.033584881 7 3 3 <action> 4 <queue> 0 192</queue></action>
</rechdr>
<recbody>
<udp>519 519 36 0</udp>
<ipv4>4 5 48 0 0 56 0 <flags>0 0 0</flags> 0 64 17 0 255.255.255.255
192.0.1.255</ipv4>
</recbody>
</rec>

<rec>
<rechdr> 3 0 0.033584881 7 3 3 <action> 5 <queue> 0 2</queue></action></rechdr>
<recbody>
<udp>519 519 36 0</udp>
<ipv4>4 5 48 0 0 56 0 <flags>0 0 0</flags> 0 64 17 0 255.255.255.255
192.0.1.255</ipv4>
</recbody>
</rec>

...
</body>
...
</trace_file>
```

FIGURE 4-203. Example of a Trace File

```
<!ELEMENT trace_file (head, body) >

<!ELEMENT head (version, scenario, comments) >

<!ELEMENT body (protocol_map*, rec*) >

<!ELEMENT version (#PCDATA) >
<!ELEMENT scenario (#PCDATA) >
<!ELEMENT comments (#PCDATA) >

<!ELEMENT protocol_map (#PCDATA) >
<!-- A protocol_map consists of the following two fields
     1) Integer corresponding to the protocol's enumeration
     2) String denoting the protocol's name
-->

<!ELEMENT rec (rechdr, recbody) >

<!ELEMENT rechdr (#PCDATA, action) >
<!-- The record header consists of the following six fields and
     an action element. Record header fields are:
     1) Originating node id
     2) Message sequence number
     3) Simulation time
     4) Originating protocol id
     5) Processing node id
     6) Tracing protocol id
-->

<!ELEMENT action (#PCDATA, queue) >
<!-- Action can have one of the following two forms
     1) Pair of integers, corresponding to action code and comment code
     2) Action code (integer) followed by a queue element
-->

<!ELEMENT queue (#PCDATA) >
(!-- Queue consists of the following two integer fields
     1) Interface id  2) Queue priority
-->

<!ELEMENT recbody (udp | tcp | ipv4 | ...)* >
<!-- The recbody element consists of protocol header elements.
     Only some of the traceable protocols are listed above.
     To add tracing for a new protocol, add it to the above list and
     define the header element corresponding to the new protocol.
 -->

<!ELEMENT udp (#PCDATA) >
<!ELEMENT ipv4 (#PCDATA | flags)* >
<!ELEMENT tcp (#PCDATA | flags)* >
<!ELEMENT flags (#PCDATA) >
```

FIGURE 4-204. Definition File for Trace File Syntax

4.9.2 Including MYPPROTOCOL in List of Traceable Protocols

When a new traceable protocol is added to EXata, it needs to be included in the list of traceable protocols. To do this, add the protocol's name to the enumeration TraceProtocolType defined in trace.h.

For our example protocol, add the entry TRACE_MYPROTOCOL to TraceProtocolType, as shown in Figure 4-205.

```
typedef enum
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,                  // 1
    TRACE_UDP,                  // 2
    TRACE_IP,                   // 3
    TRACE_CBR,                  // 4
    TRACE_FTP,                  // 5
    ...
    // Must be last one!!!
    TRACE_MYPROTOCOL,
    TRACE_ANY_PROTOCOL
}TraceProtocolType;
```

FIGURE 4-205. Adding MYPROTOCOL to List of Traceable Protocols

Note **Always add to the end of lists in header files (just before the item TRACE_ANYPROTCOL).**

4.9.3 Enabling/Disabling Tracing in Protocol's Initialization Function

Tracing for a protocol is enabled or disabled depending on the configuration parameters read from the input file. Parameter TRACE-ALL specifies whether tracing is enabled for all protocols. TRACE-ALL is read in the trace initialization function, TRACE_Initialize, which is called by the function PARTITION_InitializeNodes at the start of simulation. TRACE_Initialize is implemented in EXATA_HOME/main/trace.cpp and PARTITION_InitializeNodes is implemented in EXATA_HOME/main/partition.cpp.

The default value for TRACE-ALL is NO. If TRACE-ALL is not included in the configuration file or is included and set to NO, then tracing is disabled for all traceable protocols (unless tracing for a specific protocol is explicitly enabled, as explained below). If TRACE-ALL is included in the configuration file and is set to YES, then tracing is enabled for all traceable protocols (unless tracing for a specific protocol is explicitly disabled).

Apart from the parameter TRACE-ALL, the configuration file can specify whether tracing is enabled for a specific protocol. For example, the parameter TRACE-UDP determines whether tracing is enabled or disabled for the UDP protocol. TRACE-UDP takes precedence over TRACE-ALL, and is read in the UDP function TransportUdpInitTrace, shown in Figure 4-206. TransportUdpInitTrace is called by the UDP initialization function TransportUdpInit. TransportUdpInitTrace and TransportUdpInit are implemented EXATA_HOME/libraries/developer/src/transport_udp.cpp.

The following APIs are used enable or disable tracing for a protocol:

- TRACE_IsTraceAll: This function determines if tracing is enabled for all protocols.
- TRACE_EnableTraceXML: This function enables tracing for a specific protocol.
- TRACE_DisableTraceXML: This function disables tracing for a specific protocol.

TRACE_IsTraceAll, Trace_EnableTraceXML and TRACE_DisableTraceXML are implemented in EXATA_HOME/main/trace.cpp.

For MYPROTOCOL, define an input parameter, `TRACE-MYPROTOCOL`, which is set to YES or NO in the configuration file. In the initialization function of MYPROTOCOL, read the value of `TRACE-MYPROTOCOL` and enable or disable trace for MYPROTOCOL depending on the value of `TRACE-MYPROTOCOL` and `TRACE-ALL`. For example, function TransportUdpInitTrace calls function TRACE_EnableTraceXML if trace collection is enabled for UDP and calls function TRACE_DisableTraceXML if trace collection is disabled for UDP. The UDP function to print the UDP header, TransportUdpPrintTrace, is passed as a parameter to TRACE_EnableTraceXML. TransportUdpPrintTrace is implemented in transport_udp.cpp.

Note **For Application Layer protocols, the trace initialization function should be called from the function APP_TraceInitialize, which is implemented in EXATA_HOME/main/application.cpp.**

```
static
void TransportUdpInitTrace(Node* node, const NodeInput* nodeInput)
{
    char buf[MAX_STRING_LENGTH];
    BOOL retVal;
    BOOL traceAll = TRACE_IsTraceAll(node);
    BOOL trace = FALSE;
    static BOOL writeMap = TRUE;

    IO_ReadString(
        node->nodeId,
        ANY_ADDRESS,
        nodeInput,
        "TRACE-UDP",
        &retVal,
        buf);

    if (retVal)
    {
        if (strcmp(buf, "YES") == 0)
        {
            trace = TRUE;
        }
        else if (strcmp(buf, "NO") == 0)
        {
            trace = FALSE;
        }
        else
        {
            ERROR_ReportError(
                "TRACE-UDP should be either \"YES\" or \"NO\".\n");
        }
    }
    else
    {
        if (traceAll || node->traceData->layer[TRACE_TRANSPORT_LAYER])
        {
            trace = TRUE;
        }
    }
    if (trace)
    {
        TRACE_EnableTraceXML(node, TRACE_UDP,
            "UDP", TransportUdpPrintTrace, writeMap);
    }
    else
    {
        TRACE_DisableTraceXML(node, TRACE_UDP, "UDP", writeMap);
    }
    writeMap = FALSE;
}
```

FIGURE 4-206. Enabling/Disabling Trace Collection for UDP

4.9.4 Printing the Protocol Header

Write a function, MyprotocolPrintTrace, to print the MYPROTOCOL header. This function is called by the trace function TRACE_PrintTraceXML. Use the UDP function TransportUdpPrintTrace that prints the UDP header as a template. TransportUdpPrintTrace is shown in Figure 4-207 and is implemented in transport_udp.cpp.

```
void TransportUdpPrintTrace(Node* node, Message* msg)
{
    char buf[MAX_STRING_LENGTH];
    TransportUdpHeader* udpHdr = (TransportUdpHeader *)
                                 MESSAGE_ReturnPacket(msg);

    sprintf(buf, "<udp>%hu %hu %hu %hu</udp>",
        udpHdr->sourcePort,
        udpHdr->destPort,
        udpHdr->length,
        udpHdr->checksum);
    TRACE_WriteToBufferXML(node, buf);
}
```

FIGURE 4-207. Function to Print the UDP Header

TransportUdpPrintTrace uses the function TRACE_WriteToBufferXML to write the value of each header field to a buffer. The contents of the buffer are printed by the function TRACE_PrintTraceXML. TRACE_WriteToBufferXML and TRACE_PrintTraceXML are implemented in trace.cpp.

4.9.5 Tracing a Packet

To trace a packet, place calls to function TRACE_PrintTrace at the appropriate places in the protocol code. Usually, a trace is printed whenever a header is added or removed, a packet is dropped, or a packet is enqueued or dequeued.

Function TRACE_PrintTrace is implemented in trace.cpp. The prototype for TRACE_PrintTrace is shown below. The enumerations and `struct` definitions used below can be found in trace.h.

```
void TRACE_PrintTrace(Node* node,
                      Message* message,
                      TraceLayerType layerType,
                      PacketDirection pktDirection,
                      ActionData* actionData)
```

The parameters of TRACE_PrintTrace are:

- `node`: Pointer to the node
- `message`: Pointer to the message
- `layerType`: Layer at which the packet is being traced
- `pktDirection`: Direction of the packet: `PACKET_IN` for incoming packets, `PACKET_OUT` for outgoing packets. The direction is relative to the node, not to a specific layer.
- `actionData`: Description of the action that triggered the trace. See Section 4.9.5.1 for details.

4.9.5.1 Trace Actions

When a call to TRACE_PrintTrace is made, trace information is printed to the trace file in the form of a record (see Section 4.9.1). Each record contains information such as node and packet identifiers, simulation time, the action that triggered the trace, and the protocol headers in the packet. The data structure `ActionData`, shown below, is used to store information about the trace actions. `ActionData` and the other types that it uses are declared in trace.h.

```
typedef struct
{
    PacketActionType   actionType;
    PacketActionCommentType actionComment;
    PktQueue pktQueue;
} ActionData;
```

The fields of `ActionData` are:

- `actionType`: Action that triggered the trace. It can be one of `SEND`, `RECV`, `DROP`, `ENQUEUE` or `DEQUEUE`.
- `actionComment`: Comment giving information about the trace action. The possible comments are enumerated in `PacketActionCommentType`.

  If a new action comment is needed, add it to the enumeration `PacketActionCommentType` in trace.h.
- `pktQueue`: Details of the packet queue. The details that are printed are the interface index and priority of the queue.

4.9.5.2 Trace of a Packet Send

To trace a packet when it is sent from the node, call TRACE_PrintTrace with the appropriate parameters. An outgoing packet is usually traced after a header is added. For example, Figure 4-208 shows how an outgoing packet is traced in the UDP function TransportUdpSendToNetwork, which is implemented in transport_udp.cpp.

```
void
TransportUdpSendToNetwork(Node *node, Message *msg)
{
    TransportDataUdp *udp = (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader *udpHdr;
    AppToUdpSend *info;

    ...
    MESSAGE_AddHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);
    udpHdr = (TransportUdpHeader *) msg->packet;
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);
    udpHdr->sourcePort = info->sourcePort;
    udpHdr->destPort = info->destPort;
    udpHdr->length = (unsigned short) MESSAGE_ReturnPacketSize(msg);
    udpHdr->checksum = 0;  /* checksum not calculated */
    ActionData acnData;
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_OUT,
                     &acnData);
    NetworkIpReceivePacketFromTransportLayer(
        node,
        msg,
        info->sourceAddr,
        info->destAddr,
        info->outgoingInterface,
        info->priority,
        IPPROTO_UDP,
        FALSE,
        info->ttl);
}
```

FIGURE 4-208. Tracing an Outgoing Packet

4.9.5.3 Trace of a Packet Receive

To trace a packet when it is received at the node, call TRACE_PrintTrace with the appropriate parameters. An incoming packet is usually traced before a header is removed. For example, Figure 4-209 shows how an incoming packet is traced in the UDP function TransportUdpSendToApp, which is implemented in transport_udp.cpp.

```
void
TransportUdpSendToApp(Node *node, Message *msg)
{
    TransportDataUdp *udpLayer =
        (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader* udpHdr = (TransportUdpHeader *)
                                  MESSAGE_ReturnPacket(msg);
    ...
    ActionData acnData;
    acnData.actionType = RECV;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_IN,
                     &acnData);
    /* Remove UDP header. */
    MESSAGE_RemoveHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);
    /* Send packet to application layer. */
    MESSAGE_Send(node, msg, TRANSPORT_DELAY);
}
```

FIGURE 4-209. Tracing an Incoming Packet

4.9.5.4 Trace of a Packet Drop

To trace a packet when it is dropped at a node, call TRACE_PrintTrace with the appropriate parameters. When a packet is dropped, the trace should contain a reason for the packet drop. The reason is specified in the `actionComment` field of `actionData` (see Section 4.9.5.1). For example, Figure 4-210 shows how a packet is traced in the IP function RouteThePacketUsingLookupTable before it is dropped because no route was found. RouteThePacketUsingLookupTable is implemented in EXATA_HOME/libraries/developer/src/network_ip.cpp.

```
void
RouteThePacketUsingLookupTable(Node *node, Message *msg,
                               int incomingInterface)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;
    int outgoingInterface;
    NodeAddress nextHop;
    ...
    if (nextHop == (unsigned) NETWORK_UNREACHABLE)
    {
        ...
        ip->stats.ipOutNoRoutes++;
        //Trace drop
        ActionData acnData;
        acnData.actionType = DROP;
        acnData.actionComment = DROP_NO_ROUTE;
        TRACE_PrintTrace(node,
                         msg,
                         TRACE_NETWORK_LAYER,
                         PACKET_OUT,
                         &acnData,
                         NETWORK_IPV4);
        // Free message.
        MESSAGE_Free(node, msg);
        return;
    }
    ...
}
```

FIGURE 4-210. Tracing a Packet Drop

4.9.5.5 Trace of a Packet Enqueuing

To trace a packet when it is added to a queue, call TRACE_PrintTrace with the appropriate parameters. When a packet is enqueued, the trace should contain the interface index and the priority of the queue. For example, Figure 4-210 shows how a packet is traced in the IP function NetworkIpQueueInsert when it is added to the IP queue. NetworkIpQueueInsert is implemented in network_ip.cpp.

```
void
NetworkIpQueueInsert(
    Node *node,
    Scheduler *scheduler,
    Message *msg,
    NodeAddress nextHopAddress,
    NodeAddress destinationAddress,
    int outgoingInterface,
    int networkType,
    BOOL *queueIsFull,
    int incomingInterface,
    BOOL isOutputQueue)
{
    int queueIndex = ALL_PRIORITIES;
    IpHeaderType *ipHeader = NULL;
    ...
    //Trace Enqueue
    ActionData acn;
    acn.actionType = ENQUEUE;
    acn.actionComment = NO_COMMENT;
    NetworkType netType = NETWORK_IPV4
    acn.pktQueue.interfaceID = (unsigned short) outgoingInterface;
    acn.pktQueue.queuePriority = (
                     unsigned char) IpHeaderGetTOS(ipHeader->ip_v_hl_tos_len);
    if (outgoingInterface != CPU_INTERFACE)
    {
        TRACE_PrintTrace(node, msg, TRACE_NETWORK_LAYER, PACKET_OUT,
                         &acn, netType);
    }
    else
    {
        TRACE_PrintTrace(node, msg, TRACE_NETWORK_LAYER, PACKET_IN,
                        &acn, netType);
    }
    ...
}
```

FIGURE 4-211. Tracing a Packet Enqueue

4.9.5.6 Trace of a Packet Dequeuing

To trace a packet when it is removed from a queue, call TRACE_PrintTrace with the appropriate parameters. When a packet is dequeued, the trace should contain the interface index and the priority of the queue. For example, Figure 4-210 shows how a packet is traced in the IP function NetworkIpOutputQueueDequeuePacket when it is dequeued from the IP queue. NetworkIpOutputQueueDequeuePacket is implemented in network_ip.cpp.

```
BOOL NetworkIpOutputQueueDequeuePacket(
    Node *node,
    int interfaceIndex,
    Message **msg,
    NodeAddress *nextHopAddress,
    macHWAddress *nexthopmacAddr
    int *networkType,
    QueuePriorityType *userPriority,
    int posInQueue)
{
    BOOL dequeued = FALSE;
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    Scheduler *scheduler = NULL;
    TosType userTos = ALL_PRIORITIES;
    int outgoingInterface;
    ...
    scheduler = ip->interfaceInfo[interfaceIndex]->scheduler;
    dequeued = NetworkIpQueueDequeuePacket(node,
                                           scheduler,
                                           msg,
                                           nextHopAddress,
                                           nexthopmacAddr,
                                           &outgoingInterface,
                                           networkType,
                                           &userTos,
                                           posInQueue);
    if (dequeued)
    {
        ...
        //Trace dequeue
        ActionData acn;
        acn.actionType = DEQUEUE;
        acn.actionComment = NO_COMMENT;
        acn.pktQueue.interfaceID = (unsigned short) interfaceIndex;
        acn.pktQueue.queuePriority = (unsigned char) queuePriority;
        TRACE_PrintTrace(node, *msg, TRACE_NETWORK_LAYER, PACKET_OUT, &acn,
                         NetworkIpGetInterfaceType(node, interfaceIndex));
        ...
    }
    return dequeued;
}
```

FIGURE 4-212. Tracing a Packet Dequeue

4.10 Creating an Addon, Interface or Model Library

Libraries, Interfaces, Addons, and contributed models are different types of optional features in EXata. All are organized in a virtually identical way.

- Libraries: A library is a set of network protocols and other models for EXata. Most of the models developed by Scalable Network Technologies are organized in pre-packaged libraries. Source code and other files for libraries are located in EXATA_HOME/libraries.
- Interfaces: An interface typically implements an instance of EXata's external interface API to allow EXata to inter-operate with some third party software. An interface module typically requires third party software for compilation and use. For example, use of the HLA interface requires the user to install an HLA RTI. Source code and other files for interfaces are located in EXATA_HOME/interfaces.
- Contributed Models: These models are often individual protocols or models developed and contributed by a EXata user for distribution within the user community. Source code and other files for contributed models are located in EXATA_HOME/contributed.
- Addons: These are special-purpose custom modules or prototypes of models under development. Source code and other files for addons are located in EXATA_HOME/addons.

For the sake of brevity, we will use the term "library" in the remainder of this section to refer to all four classes of optional features.

We illustrate the process of adding a library to EXata by creating a sample library, the user_models library. For simplicity, the user_models library adds an Application Layer protocol, HELLO, but a programmer can develop a library that adds functionality at any other layer or at multiple layers.

The following list summarizes the actions that need to be performed for adding a traffic-generating Application Layer protocol, HELLO, to the user-created library, user_models. Each of these steps is described in detail in subsequent sections. Although a traffic-generating Application Layer protocol typically has two components, a client and a server, for simplicity, we describe the process for only one component for HELLO, instead of for both client and server.

1. Create a directory and header and source files (see Section 4.10.1).
2. Include the protocol in the list of Application Layer protocols (see Section 4.10.2).
3. Develop components of the protocol (see Section 4.10.3).
4. Modify the file application.cpp to include the protocol's header file (see Section 4.10.4).
5. Make calls to the model's functions from the Application Layer functions (see Section 4.10.4).
6. Integrate the library into EXata (see Section 4.10.5).

Note **It is desirable to make libraries modular so that individual libraries can be included or excluded from EXata easily and without changes to code external to the library. To enable this, all changes made to code not in your library should be made conditional on a compilation switch. For the user_models library, the compilation switch is USER_MODELS_LIB. This switch is defined as the value of the variable USER_MODELS_OPTIONS in EXATA_HOME/libraries/user_models/Makefile-common (see Section 4.10.5.1).**

4.10.1 Creating Directory and Files

Create a directory in EXATA_HOME/libraries called user_models. Create a subdirectory in EXATA_HOME/libraries/user_models called src. The Makefiles for the library will be placed in the top directory, EXATA_HOME/libraries/user_models. The code for the application model will be put in EXATA_HOME/libraries/user_models/src.

Create header and source files for the application model in the directory EXATA_HOME/libraries/user_models/src. The example application model, HELLO, has only one header and one source file. But a model can have multiple header files and multiple source files. If the model adds functionality at multiple layers, it is recommended to have separate header and source files for each layer. See Section 4.2.5.2 for details of contents of the header and source files for a traffic-generating Application Layer protocol.

Figure 4-213 shows an example of the header file for the HELLO application, app_hello.h.

```
#ifndef HELLO_H
#define HELLO_H
...
void AppHelloInit(Node *node, NodeInput *nodeInput);
void AppHelloProcessEvent(Node *node, Message *packet);
void AppHelloFinalize(Node *node);
#endif
```

FIGURE 4-213. Header File for HELLO Application

In the above example, AppHelloInit, AppHelloProcessEvent and AppHelloFinalize are the initialization, event handler and finalization functions for HELLO, respectively.

Figure 4-214 shows an example of the source file for the HELLO application, app_hello.cpp.

```
#include <stdlib.h>
#include "api.h"
#include "app_hello.h"
void AppHelloInit(Node *node, const NodeInput *nodeInput)
{
     printf("AppHelloInit called.\n");
}
void AppHelloProcessEvent(Node *node, Message *packet)
{
     printf("AppHelloProcessEvent called.\n");
}
void AppHelloFinalize(Node *node)
{
     printf("AppHelloFinalize called.\n");
}
```

FIGURE 4-214. Source File for HELLO Application

4.10.2 Including HELLO in List of Application Layer Protocols

This step is identical to including an Application Layer protocol as discussed in Section 4.2.5.3.

For our example protocol, add the entry APP_HELLO to AppType, as shown in Figure 4-215.

Note **Add the new application at the end of the list because the items in this enumeration are used to initialize random variables, which must not depend on the inclusion of this addon.**

```
typedef enum
{
    APP_FTP_SERVER_DATA = 20,
    APP_FTP_SERVER = 21,
    APP_FTP_CLIENT,
    ...
 #ifdef USER_MODELS_LIB
    APP_HELLO,
#endif USER_MODELS_LIB
    APP_PLACEHOLDER
} AppType;
```

FIGURE 4-215. Adding HELLO to List of Application Layer Protocols

4.10.3 Developing Protocol Components

The following list summarizes the actions that need to be performed for developing components of an Application Layer protocol.

1. Declare data structures for the protocol, as described in Section 4.2.5.4, in the header file app_hello.h.
2. Write the initialization function for the protocol, as described in Section 4.2.5.5.
3. Write the event dispatcher function for the protocol, as described in Section 4.2.5.6.2.
4. Write the finalization function for the protocol, as described in Section 4.2.5.8.2.
5. Write the other implementation functions for the protocol.
6. Implement code for collecting and printing statistics for the protocol, as described in Section 4.2.5.7.
7. Include all functions in the source file, app_hello.cpp. Include the prototypes of all interface functions in the header file, app_hello.h.

4.10.4 Calling Protocol Functions from Application Layer Functions

The initialization, event dispatcher, and finalization functions for the HELLO application are called from the Application Layer initialization, event dispatcher, and finalization functions, in a manner similar to adding an Application Layer protocol, described in Section 4.2.5, except that for HELLO, these calls are conditional on the user_models library being activated.

1. Include the header file, app_hello.h, in EXATA_HOME/main/application.cpp, as shown in Figure 4-216.

```
...
#ifdef ADDON_LINK16
#include "link16_cbr.h"
#endif // ADDON_LINK16

#ifdef ADDON_USER_MODELS
#include "app_hello.h"
#endif /* ADDON_USER_MODELS */

...
```

FIGURE 4-216. Including HELLO Header File in application.cpp

2. Call the protocol's initialization function, AppHelloInit, from the Application Layer initialization function APP_InitializeApplications, as shown in Figure 4-217. Function APP_InitializeApplications is defined in EXATA_HOME/main/application.cpp. Read and store the configuration parameters for HELLO in APP_InitializeApplications. See Section 4.2.5.5.2 for details of reading configuration parameters from an input file.

```
void APP_InitializeApplications(Node *firstnode, const NodeInput *nodeInput)
{
    ...
    for (i = 0; i < appInput.numLines; i++)
    {
        sscanf(appInput.inputStrings[i], "%s", appStr);
        ...
        if (strcmp(appStr, "FTP") == 0)
        {
           ...
        }
        ...
        else
        if (strcmp(appStr, "mgen") == 0)
        {
#ifdef ADDON_MGEN4
            ...
#endif // ADDON_MGEN4
        }
         else
         if (strmcp(appStr, "HELLO") == 0)
         {
#ifdef USER_MODELS_LIB
            ...
            /*Read user input into appropriate variables */
            /* Call HELLO initialization function */
            AppHelloInit (node, nodeInput);
#endif // USER_MODELS_LIB
         }
         else
         ...
    }
    ...
}
```

FIGURE 4-217. Calling HELLO Initialization Function

3. Call the protocol's event handler function from the Application Layer event handler function, APP_ProcessEvent, as shown in Figure 4-218. Function APP_ProcessEvent is defined in EXATA_HOME/main/application.cpp.

```
void APP_ProcessEvent(Node *node, Message *msg)
{
    short protocolType;
    protocolType = APP_GetProtocolType(node,msg);
    switch(protocolType)
    {
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
        ...
 #ifdef USER_MODELS_LIB
        case APP_HELLO:
        {
            AppHelloProcessEvent(node, msg);
            break;
        }
 #endif /* USER_MODELS_LIB */
        ...
 }//switch//
}
```

FIGURE 4-218. Calling HELLO Event Dispatcher Function

4. Call the protocol's finalization function from the Application Layer finalization function, APP_Finalize, as shown in Figure 4-219. Function APP_Finalize is defined in EXATA_HOME/main/application.cpp.

```
void App_Finalize (Node *node)
{
  ...
  AppInfo *applist = NULL;
  AppInfo *nextApp = Null;
  ...
  for (appList = node->appData.appPtr; appList != NULL;
       appList = nextApp)
    {
        switch (appList->appType)
        {
         ...
         case APP_CBR_CLIENT:
            {
                AppCbrClientFinalize(node, appList);
                break;
            }
          ...
 #ifdef USER_MODELS_LIB
          case APP_HELLO
            {
                AppHelloFinalize(node, appList);
                break;
            }
 #endif /* USER_MODELS_LIB */
             ...
         nextApp = appList->appNext;
    }
  ...
}
```

FIGURE 4-219. Calling HELLO Finalization Function

4.10.5 Integrating a New Library into EXata

To integrate your library into EXata, create Makefiles for your library, as described in Section 4.10.5.1, and include the library Makefile for your platform in the main Makefile, as described in Section 4.10.5.2. Recompile EXata, as described in Section 4.10.5.3.

4.10.5.1 Creating Makefiles

In the directory EXATA_HOME/libraries/user_models, create a file Makefile-common that specifies the source files to be included and any other platform-independent information required for compilation. Create a file Makefile-windows (for Windows platforms) or Makefile-unix (used for Linux platforms). Platform-specific information is included in these two files. The files to be added are shown below.

1. File EXATA_HOME/libraries/user_models/Makefile-common:

```
USER_MODELS_OPTIONS = -DUSER_MODELS_LIB
USER_MODELS_DIR = ../libraries/user_models/src
USER_MODELS_SRCS = \
$(USER_MODELS_DIR)/app_hello.cpp
USER_MODELS_INCLUDES = \
-I$(USER_MODELS_DIR)
```

2. File EXATA_HOME/libraries/user_models/Makefile-windows:

```
include ../libraries/user_models/Makefile-common
ADDON_OPTIONS    = $(ADDON_OPTIONS) $(USER_MODELS_OPTIONS)
ADDON_SRCS       = $(ADDON_SRCS) $(USER_MODELS_SRCS)
ADDON_INCLUDES   = $(ADDON_INCLUDES) $(USER_MODELS_INCLUDES)
```

3. File EXATA_HOME/libraries/user_models/Makefile-unix:

```
include ../libraries/user_models/Makefile-common
ADDON_OPTIONS    += $(USER_MODELS_OPTIONS)
ADDON_SRCS       += $(USER_MODELS_SRCS)
ADDON_INCLUDES   += $(USER_MODELS_INCLUDES)
```

4.10.5.2 Include Library Makefile in Main Makefile

Enable the library by including its Makefile in the addons Makefile for your platform (Makefile-addons-windows or Makefile-addons-unix).

For Windows, make the following entry in the file EXATA_HOME/main/Makefile-addons-windows:

```
...
# INSERT LIBRARIES HERE...
# USER_MODELS library
include ../libraries/user_models/Makefile-windows
#
...
```

For Linux, make the following entry in the file EXATA_HOME/main/Makefile-addons-unix:

```
...
# INSERT LIBRARIES HERE...
# USER_MODELS library
include ../libraries/user_models/Makefile-unix
#
...
```

4.10.5.3 Recompiling EXata

After creating and modifying the Makefiles, recompile EXata.

Note **To correctly integrate a library into EXata, you must delete all object files before recompiling.**

Windows

Use the following commands to remove all object (.obj) files and recompile:

```
nmake clean
nmake
```

See Section 2.2 for detailed instructions for recompiling EXata on Windows.

Linux

Use the following commands to remove all object (.o) files and recompile:

```
make clean
make
```

See Section 2.3 for detailed instructions for recompiling EXata on Linux.

4.11 Communication Between Layers

Although EXata protocols follow a strict, adjacent layering approach, protocols can be written to deviate from this and communicate across layers and even across nodes.

4.11.1 Communication Between Adjacent Layers

This section covers the general approach used to communicate between adjacent layers.

Communication between adjacent layers is accomplished either by using the message API MESSAGE_Send (see Section 3.3.1.2) or by using direct function calls. A protocol may use MESSAGE_Send directly (see Section 3.3.2.1.2) or it can use one of the layer-specific APIs which are implemented using MESSAGE_Send (see Section 3.3.2.1.1).

As an example, an application that relies on UDP may use MESSAGE_Send directly, or it may call one of the APIs listed in Table to pass data to UDP. Figure 4-220 shows the implementation of one of these functions, APP_UdpSendNewData, with comments added for explication. APP_UdpSendNewData sends application data to UDP using MESSAGE_Send, and is implemented in app_util.cpp.

```
void
APP_UdpSendNewData(
    Node *node,
    AppType appType,
    NodeAddress sourceAddr,
    short sourcePort,
    NodeAddress destAddr,
    char *payload,
    int payloadSize,
    clocktype delay,
    TraceProtocolType traceProtocol)
{
    Message *msg;
    AppToUdpSend *info;
    ActionData acnData;

    // Create a packet event with the Transport Layer as the destination layer
    // and UDP as the destination protocol. The event type is
    // MSG_TRANSPORT_FromAppSend.
    msg = MESSAGE_Alloc(
              node,
              TRANSPORT_LAYER,
              TransportProtocol_UDP,
              MSG_TRANSPORT_FromAppSend);

    // ALlocate memory to store data.
    MESSAGE_PacketAlloc(node, msg, payloadSize, traceProtocol);

    // Copy application data into memory just allocated.
    memcpy(MESSAGE_ReturnPacket(msg), payload, payloadSize);

    // Create and assign info field values for UDP.
    MESSAGE_InfoAlloc(node, msg, sizeof(AppToUdpSend));
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);

    SetIPv4AddressInfo(&info->sourceAddr, sourceAddr);
    info->sourcePort = sourcePort;

    SetIPv4AddressInfo(&info->destAddr, destAddr);
    info->destPort = (short) appType;
    info->priority = APP_DEFAULT_TOS;
    info->outgoingInterface = ANY_INTERFACE;
    info->ttl = IPDEFTTL;
    ...
    // Send the message to UDP at the Transport Layer.
    MESSAGE_Send(node, msg, delay);
}
```

FIGURE 4-220. Communication Between Layers Using MESSAGE_Send

The other method of passing data between adjacent layers is to use direct function calls. An example of this is communication between UDP at the Transport Layer and IP at the Network Layer. UDP function TransportUdpSendToNetwork appends a UDP header to the user data received from an application running at the Application Layer and sends the resulting packet to IP using the function NetworkIpReceivePacketFromTransportLayer. TransportUdpSendToNetwork is shown in Figure 4-221 and is implemented in EXATA_HOME/libraries/developer/src/transport_udp.cpp. Note that MESSAGE_Send is

not used in this case. NetworkIpReceivePacketFromTransportLayer is implemented in EXATA_HOME/libraries/developer/src/network_ip.cpp.

```
void
TransportUdpSendToNetwork(Node *node, Message *msg)
{
    TransportDataUdp *udp = (TransportDataUdp *) node->transportData.udp;
    TransportUdpHeader *udpHdr;
    AppToUdpSend *info;

    if (udp->udpStatsEnabled == TRUE)
    {
        udp->statistics->numPktFromApp++;
    }

    MESSAGE_AddHeader(node, msg, sizeof(TransportUdpHeader), TRACE_UDP);

    udpHdr = (TransportUdpHeader *) msg->packet;
    info = (AppToUdpSend *) MESSAGE_ReturnInfo(msg);

    udpHdr->sourcePort = info->sourcePort;
    udpHdr->destPort = info->destPort;
    udpHdr->length = (unsigned short) MESSAGE_ReturnPacketSize(msg);
    udpHdr->checksum = 0;  /* checksum not calculated */

    ActionData acnData;
    acnData.actionType = SEND;
    acnData.actionComment = NO_COMMENT;
    TRACE_PrintTrace(node,
                     msg,
                     TRACE_TRANSPORT_LAYER,
                     PACKET_OUT,
                     &acnData);

    NetworkIpReceivePacketFromTransportLayer(
        node,
        msg,
        info->sourceAddr,
        info->destAddr,
        info->outgoingInterface,
        info->priority,
        IPPROTO_UDP,
        FALSE,
        info->ttl);
}
```

FIGURE 4-221. Communication Between Layers Using Function Calls

The method of layer communication (whether to use direct function calls or MESSAGE_Send) is dependent on the layer and how it is implemented. It should be noted that the simulator's performance is better if direct function calls are used, as compared to using MESSAGE_Send. This is because function calls bypass the event scheduling system of EXata, and thus some overhead is reduced. However, direct function calls can not model delays, and MESSAGE_Send must be used if delays are required to be modeled between layers.

4.11.2 Communication Between Non-adjacent Layers

Although the design philosophy of EXata is for adjacent layers to communicate with each other, it is possible for protocols at non-adjacent layers to communicate with each other. For instance, protocols written at the Application Layer, if desired, may bypass the Transport Layer and communicate directly with the Network Layer. This procedure would entail the Application Layer protocol to use the Network Layer APIs instead of the Transport Layer APIs.

As an example, suppose we want the current CBR application to bypass the Transport Layer protocol (UDP in this case) and communicate directly with the Network Layer. Also, suppose that we want to maintain the current Network Layer APIs. In order to achieve this, we must close the gap between the Application and Network Layer as shown in Figure 4-222.

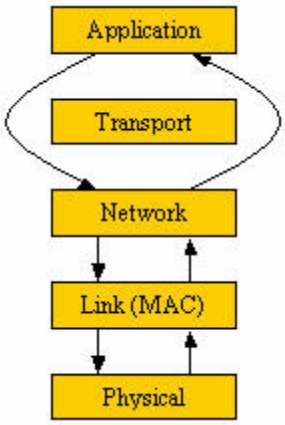


FIGURE 4-222. Bypassing the Transport Layer

Section 4.11.2.1 describes the communication from the Application Layer to the Network Layer, and Section 4.11.2.2 describes the communication from the Network Layer to the Application layer, bypassing the Transport Layer in each case.

4.11.2.1 Application Layer to Network Layer Communication

Normally, a UDP-based application sends user data to UDP, which appends a UDP header to the packet and sends it to IP at the Network Layer. To enable CBR to directly send user data to IP, we have to first determine what APIs are used by UDP to communicate with the Network Layer, and then use the same interface between the CBR and IP. In this case, CBR calls API function APP_UdpSendNewHeaderVirtualDataWithPriority to send data to UDP, and UDP uses function NetworkIpReceivePacketFromTransportLayer to pass data to the Network Layer. Therefore, the modified CBR application should replace the call to APP_UdpSendNewHeaderVirtualDataWithPriority with a call to NetworkIpReceivePacketFromTransportLayer, with appropriate parameters. Additionally, any actions performed in APP_UdpSendNewHeaderVirtualDataWithPriority should be performed by CBR directly before calling NetworkIpReceivePacketFromTransportLayer.

CBR function AppLayerCbrClient, shown in Figure 4-25, calls the API function APP_UdpSendNewHeaderVirtualDataWithPriority to send data to UDP. To enable CBR to send data to IP directly, replace the call to APP_UdpSendNewHeaderVirtualDataWithPriority with a call to NetworkIpReceivePacketFromTransportLayer, as shown in Figure 4-223. CBR functions are implemented in EXATA_HOME/libraries/developer/src/app_cbr.cpp and function NetworkIpReceivePacketFromTransportLayer is implemented in network_ip.cpp.

APP_UdpSendNewHeaderVirtualDataWithPriority is implemented in EXATA_HOME/main/app_util.cpp, and prototypes for message APIs can be found in the file EXATA_HOME/include/message.h.

The steps needed to enable CBR to directly communicate with IP are listed below.

1. To enable CBR to call the IP function NetworkIpReceivePacketFromTransportLayer, include the file network_ip.h in the file app_cbr.cpp by inserting the following statement:

```
#include "network_ip.h"
```

2. When IP receives a packet from the upper layers, it appends an IP header to the packet. One of the fields of the IP header is the identifier of the protocol to which IP delivers the packet at the destination. To enable IP to deliver a packet to CBR, define an identifier for CBR by including the following statement in EXATA_HOME/libraries/developer/src/network_ip.h:

```
#define IPPROTO_CBR             255
```

   Here, 255 is used as an example. Be sure to use a number that is not used by any other protocol.
3. In CBR function AppLayerCbrClient, declare a new message variable `newMsg` and allocate memory for it using API function MESSAGE_Alloc (see Figure 4-223). This message variable is used only to send data to IP and not to schedule an event. Therefore the layer, protocol and event type of the message are not important and are set to 0.
4. Copy the CBR data into the `packet` field of `newMsg` using the function memcpy (see Figure 4-223).
5. Update the `virtualPayLoadSize` field of `newMsg` (see Section 3.3.1.1) by using the API function MESSAGE_AddVirtualPayload (see Figure 4-223). This is normally done in the function APP_UdpSendNewHeaderVirtualDataWithPriority, but since that API function is not being used in this modification, `virtualPayLoadSize` field of the message should be updated in AppLayerCbrClient before data is sent to IP. See Section 4.2.5.6.2 for an explanation of the use of `virtualPayLoadSize` field,
6. Send the CBR data to IP by calling function NetworkIpReceivePacketFromTransportLayer (see Figure 4-223). In the function call, constant IPDEFTTL is the default value of the TTl field.

```
void AppLayerCbrClient(Node *node, Message *msg)
{
    ...
    switch(msg->eventType)
    {
        case MSG_APP_TimerExpired:
        {
            ...
            switch (timer->type)
            {
                case APP_TIMER_SEND_PKT:
                {
                    CbrData data;
                    Message *newMsg;
                    ...
                    // Create a new message to hold the data.
                    newMsg = MESSAGE_Alloc(node, 0, 0, 0);
                   // Allocate memory for user data.
                   MESSAGE_PacketAlloc(node, newMsg, sizeof(data), TRACE_CBR);
                   // Copy user data into packet field.
                   memcpy(MESSAGE_ReturnPacket(newMsg), &data, sizeof(data));
                   // Update virtualPayLoadSize field of message.
                   MESSAGE_AddVirtualPayload(node,
                                             newMsg,
                                          clientPtr->itemSize - sizeof(data));
                   // Send message to IP.
                   NetworkIpReceivePacketFromTransportLayer(
                                          node,
                                          newMsg,
                                          clientPtr->localAddr,
                                          clientPtr->remoteAddr,
                                          ANY_INTERFACE,
                                          clientPtr->tos,
                                          IPPROTO_CBR,
                                          FALSE,
                                          IPDEFTTL);
                     clientPtr->numBytesSent += clientPtr->itemSize;
                     ...
                  }
               ...
            }
            break;
        }
         ...
    }
    MESSAGE_Free(node, msg);
}
```

FIGURE 4-223. Application Layer to Network Layer Bypassing Transport Layer

4.11.2.2 Network Layer to Application Layer Communication

For communication from the Network Layer to the Application Layer, first determine what APIs are used between the Transport and Application Layers and then have the Network Layer use the same interface to pass data directly to the Application Layer.

At the Network Layer, IP function DeliverPacket reads the protocol number for the destination protocol of a received packet from the packet's IP header and passes the packet to the destination protocol. For a UDP-based application, DeliverPacket sends the received packet to UDP using the function SendToUdp. UDP, in turn, passes the packet to the application using the UDP function TransportUdpSendToApp. The IP functions DeliverPacket and SendToUdp are implemented in network_ip.cpp. The UDP function TransportUdpSendToApp is shown in Figure 4-57 and is implemented in transport_udp.cpp.

To enable IP to send data to CBR directly, write a function SendToCbr and modify function DeliverPacket to call SendToCbr to deliver packets to CBR. The steps needed to enable IP to directly communicate with CBRT are listed below.

1. Define a data structure `IpToAppRecv`, similar to the data structure `UdpToAppRecv`, in EXATA_HOME/include/api.h, as shown below:

```
struct IpToAppRecv
{
    Address sourceAddr;
    short sourcePort;
    Address destAddr;
    short destPort;
    int incomingInterfaceIndex;
};
```

   This data structure is used for the `info` field of a message for communication between IP and the Application Layer.

2. Modify the IP function DeliverPacket by adding a case for CBR in the second switch statement, as shown in Figure 4-224. The constant IPPROTO_CBR is explained in Section 4.11.2.1.

```
static void
DeliverPacket(Node *node, Message *msg,
              int interfaceIndex, NodeAddress previousHopAddress)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    NodeAddress sourceAddress = 0;
    NodeAddress destinationAddress =0;
    unsigned char ipProtocolNumber;
    unsigned ttl =0;
    TosType priority;
    ...
    IpHeaderType *ipHeader = (IpHeaderType *) msg->packet;
    ...
    ipProtocolNumber = ipHeader->ip_p;
    ...
    switch (ipProtocolNumber)
    {
        // Delivery to Transport Layer protocols.

        case IPPROTO_UDP:
        {
            SendToUdp(node, msg, priority, sourceAddress, destinationAddress,
                      interfaceIndex);
            break;
        }
        ...
        case IPPROTO_CBR:
        {
            SendToCbr(node, msg, priority, sourceAddress,
                      destinationAddress, interfaceIndex);
            break;
        }
        ...
    }
    ...
}
```

FIGURE 4-224. Network Layer to Application Layer Bypassing Transport Layer

3. Write the function SendToCbr in network_ip.cpp, as shown in Figure 4-225.

   Function SendToCbr uses the event type `MSG_APP_FromTransport` to send a packet from IP to the Application Layer. This name is misleading because the packet is being sent from the Network Layer, not from the Transport Layer. A more meaningful name for the event type would be `MSG_APP_FromNetwork`. If this event type is used, declare the new event type in api.h, as described in Section 4.2.5.6.2, and in the CBR server event dispatcher function AppLayerCbrServer, shown in Figure 4-26, replace `MSG_APP_FromTransport` with `MSG_APP_FromNetwork`.

```
void SendToCbr(
     Node *node,
     Message *msg,
     TosType priority,
     NodeAddress sourceAddress,
     NodeAddress destinationAddress,
     int incomingInterfaceIndex)
{
     CbrData data;
     IpToAppRecv *info;

     // Get the CBR header to get source port information.
     memcpy (&data, MESSAGE_ReturnPacket(msg), sizeof(data));

    // Set layer and protocol to Application Layer and CBR Server respectively.
     MESSAGE_SetLayer(msg, APP_LAYER, APP_CBR_SERVER);

     // Set event type to one recognized by CBR.
     MESSAGE_SetEvent(msg, MSG_APP_FromTransport);

     // Set the info field (this will be used by CBR Server).
     MESSAGE_InfoAlloc(node, msg, sizeof(IpToAppRecv));
     info = (IpToAppRecv *) MESSAGE_ReturnInfo(msg);
     SetIPv4AddressInfo(&(info->sourceAddr), sourceAddress);
     info->sourcePort = data.sourcePort;
     SetIPv4AddressInfo(&(info->destAddr), destinationAddress);
     info->destPort = APP_CBR_SERVER;
     info->incomingInterfaceIndex = incomingInterfaceIndex;

    // Send packet to Application Layer.
    MESSAGE_Send(node, msg, PROCESS_IMMEDIATELY);
}
```

FIGURE 4-225. Delivering Packets from IP to CBR

4.11.3 Communication Among Layers Across Nodes

In addition to supporting the ability to communicate between non-adjacent layers, EXata also allows the capability to bypass the protocol stack and communicate directly across nodes. For instance, an application protocol at one node can directly communicate with an application protocol at another node. This is not restricted to just between the same applications. Cross-node communication can also occur between different protocols in the same layer or of different layers.

Note: In general, we do not recommend using this capability. Bypassing the lower layers of the protocol stack may result in misleading statistical results. Use of this capability also has an impact on parallel execution, including disabling some optimizations that are implemented at the lower layers of the stack. However, if handled properly, it can result in improved runtime performance.

To continue with our CBR example, if we wanted to model only a fixed delay when sending CBR traffic between the source node and the destination node, we can achieve this by sending a message directly from the CBR source node to the CBR destination node. To achieve this, modify the CBR function AppLayerCBrClient, as shown in Figure 4-226. AppLayerCbrClient is implemented in the file EXATA_HOME/libraries/developer/src/app_cbr.cpp.

Also, include the file partition.h in app_cbr.cpp to access the partition data structure, which is required to map the destination node identifier to a node pointer, by inserting the following statement in app_cbr.cpp:

```
#include "partition.h"
```

The comments in Figure 4-226 explain the purpose of each line of code.

```
void AppLayerCbrClient(Node *node, Message *msg)
{
    ...
            switch (timer->type)
            {
                case APP_TIMER_SEND_PKT:
                {
                    CbrData data;
                    Message *newMsg;
                    Node *destNode;
                    NodeAddress destId;
                    UdpToAppRecv *info;
                    ...
                    // Get the destination nodeId.
                    destId = MAPPING_GetNodeIdFromInterfaceAddress(
                                node, GetIPv4Address(clientPtr->remoteAddr));
                   // nodeIsLocal tells whether the destination node exists on
                    // the same processor as the current node.
                    BOOL nodeIsLocal = FALSE;
                    // Get the destination node pointer.
                    nodeIsLocal = PARTITION_ReturnNodePOinter(
                                          node->partitionData,  &destNode,
                                          destId, TRUE);
                    // Make sure that the destination node pointer exists.
                    assert (destNode != NULL);
                  // Create a new message to send to CBR Server at destination.
                    newMsg = MESSAGE_Alloc(node, APP_LAYER, APP_CBR_SERVER,
                                          MSG_APP_FromTransport);
                   // Allocate memory for user data.
                   MESSAGE_PacketAlloc(node, newMsg, sizeof(data), TRACE_CBR);
                   memcpy(MESSAGE_returnPacket(newMsg), &data, sizeof(data));
                   // Create and set the info field.
                   MESSAGE_InfoAlloc(node, newMsg, sizeof(UdpToAppRecv));
                   info = (UdpToAppRecv *) MESSAGE_ReturnInfo(msg);
                   info->sourceAddr = clientPtr->localAddress;
                   info->sourcePort = clientPtr->sourcePort;
                   info->destAddr = clientPtr->remoteAddress;
                   info->destPort = APP_CBR_SERVER;
                   info->incomingInterfaceIndex = 0;
                  // Send the message to the destination node after 1 ms delay.
                   if (nodeIsLocal){
                      MESSAGE_Send(destNode, newMsg, 1 * MILLI_SECOND);
                   } else {
                       // If the node is not local, ise MESSAGE_RemoteSend
                       // instead of MESSAGE_Send to deliver the message.
                   MESSAGE_RemoteSend(destNode, destId, newMsg,
                                      1 * MILLI_SECOND);
                   }
                   clientPtr->numBytesSent += clientPtr->itemSize;
        ...
}
```

FIGURE 4-226. Code Sample to Bypass Layers and Communicate Between Nodes

In addition to the change to the layer function, the user must specify a minimum lookahead (i.e., delay) on the message in order to enable parallel execution. Assuming the minimum delay is 1 millisecond, the following code segment must be added to the CBR_ClientInit function.

```
#ifdef PARALLEL
    PARALLEL_SetProtocolIsNotEOTCapable(node);
    PARALLEL_SetMinimumLookaheadForInterface(
        node,
        (1 * MILLISECONDS));
#endif // PARALLEL
```

5 Customizing EXata Graphical User Interface (GUI)

The chapter describes how to add capabilities to the EXata Graphical User Interface (GUI).

The GUI components in EXata include the following:

- EXata Architect: Used to create and design experiments (in Design mode) and to execute and animate experiments (in Visualize mode).
- EXata Analyzer: Used for displaying graphs of collected statistics.
- EXata Packet Tracer: Used for displaying packet traces.

These tools in EXata allow for a limited amount of customization. While the source code is not available to users, settings files in the EXATA_HOME/gui/settings directory allow the tools to be customized.

5.1 Customizing Design Mode of EXata Architect

Details for using EXata Architect are provided in *EXata User's Guide*. This section describes how to customize the Design mode of EXata Architect.

Chapter 4 describes how to develop code for new models at different layers of the protocol stack. This section describes how to integrate new models into Architect by modifying some settings files. Section 5.1.1 provides an overview of these settings files. Section 5.1.2 gives a detailed description of the elements used in these files and Section 5.1.3 describes the interaction among these files. Section 5.1.4 explains how to modify these settings files to integrate a new protocol into Architect.

5.1.1 Description of EXata GUI Settings Files

EXata Architect provides property editors for a user to input scenario and protocol parameters. These parameters are used to create the EXata configuration files (.config, .app, etc.), which are used as input files for EXata Simulator. There is a property editor associated with each scenario component, such as a device, link, application, or network object. In addition, there are property editors for setting global properties and interface properties.

In a property editor, related parameters are grouped together. Each group of parameters appears under a tab or as a list item under a tab. In this document, we refer to such a related group of parameters as a *segment* of a property editor. A segment may be a part of several property editors and may appear at different places in different property editors. For example, consider the property editors for the default device (see Figure 5-1) and the wireless subnet (see Figure 5-2). The Routing Protocol segment, which groups together routing-related parameters, appears in both property editors: as a list item in one and as a tab in the other.

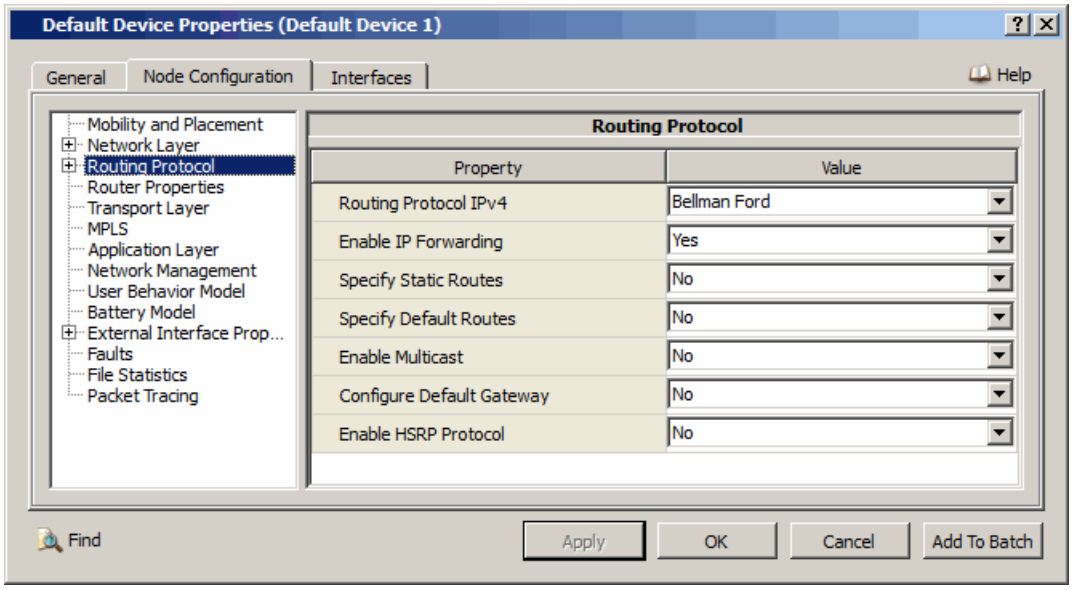


FIGURE 5-1. Default Device Property Editor

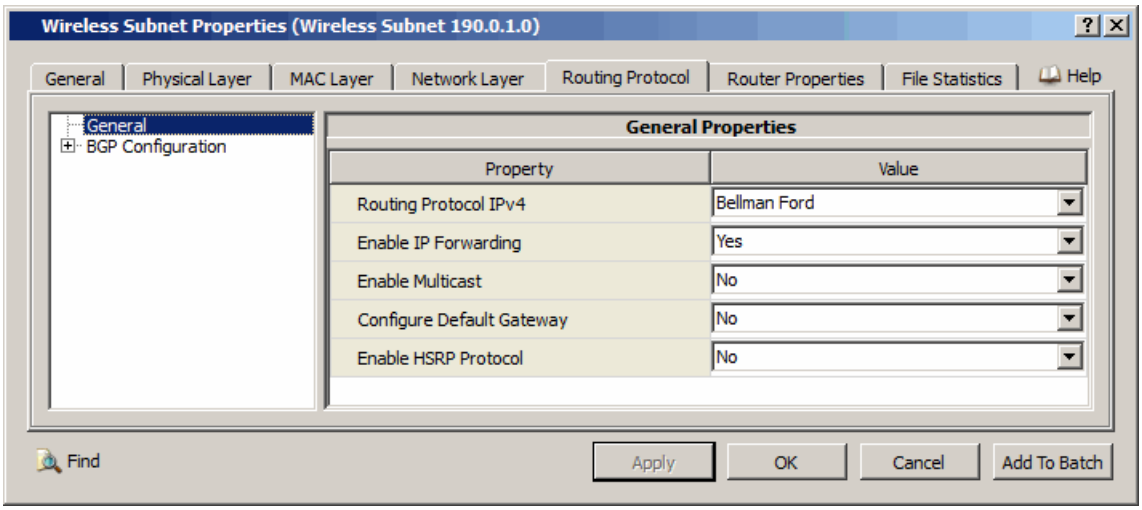


FIGURE 5-2. Wireless Subnet Property Editor

EXata Architect uses several settings files to build the property editors. This section explains the general structure and usage of these files, but is not a complete reference. These files are of three types:

- **Component Files**: A component file is associated with each property editor in Architect and contains structural information for displaying all segments of the property editor. For each segment of the property editor, the component file contains its detailed description or has a reference to a shared description (see Shared Description Files below). All component files have the extension ".cmp" and are stored in the folder EXATA_HOME/gui/settings/components.
- **Shared Description Files**: Many segments are common to several property editors. To allow sharing of descriptions of these common segments, the segment descriptions are put in shared description files. Descriptions of related segments are contained in the same shared description file. Shared description files have the extension ".prt" and are stored in the folder EXATA_HOME/gui/settings/protocol_models. Section 5.1.3 describes how component files use descriptions contained in shared description files.
- **Toolset Description File**: The file standard.xml contains the description of the layout of Architect Toolset, i.e., the icons for each device type, application, link, etc. This file is stored in the folder EXATA_HOME/gui/settings/Toolsets.

5.1.1.1 Structure of GUI Settings Files

The GUI settings files are in standard XML format and have a common structure. These files can contain the following four nested elements, each of which is composed of named attributes:

- `category`: The `category` element is the top level element and can have `subcategory` and `variable` elements as its children. In a component file, the `category` element represents the property editor. In a shared description file, the `category` element represents the group of shared segment descriptions. See Section 5.1.2.1 for the attributes of the `category` element.
- `subcategory`: The `subcategory` element can have `subcategory` and `variable` elements as its children. The `subcategory` element contains information for displaying a segment in a property editor. See Section 5.1.2.2 for the attributes of the `subcategory` element.
- `variable`: The `variable` element can have `variable` and `option` elements as its children. The `variable` element corresponds to a parameter in a property editor. See Section 5.1.2.3 for the attributes of the `variable` element.
- `option`: The `option` element can have `variable` elements as its children. If a parameter is of type list (i.e., it can take as its value one of an enumerated list of values), the `option` element corresponds to an enumeration of the list. See Section 5.1.2.4 for the attributes of the `option` element.

5.1.1.2 Component Files

In Architect, there is a property editor for each of the components in the Standard Toolset of Architect. In addition, there are property editors for global (scenario) parameters, for interface parameters, for ATM link parameters, and for ATM interface parameters.

Table 5-1 lists the different property editors, the component file that describes the property editor, and the configuration file generated by the component file. The component files are stored in EXATA_HOME/gui/settings/components.

TABLE 5-1. Property Editors and Component Files

Property Editor	Component File	Configuration File Generated
Antenna (embedded in Antenna Model Editor)	antenna.cmp	.antenna-models
ATM	atm.cmp	.config

TABLE 5-1. Property Editors and Component Files (Continued)

Property Editor	Component File	Configuration File Generated
ATM Interface	atminetrface.cmp	.config
ATM Link	atmlink.cmp	.config
BGP Link	bgp.cmp	.bgp
Default Device	defaultnode.cmp	.config
Hierarchy	hierarchy.cmp	.config
Interface	interface.cmp	.config
Point-to-point Link	point_to_point_link.cmp	.config
Satellite	satellite.cmp	.config
Scenario (Global Settings)	scenario.cmp	.config
Switch	switch.cmp	.config
Wired Subnet	subnet.cmp	.config
Wireless Subnet	wireless-subnet.cmp	.config
CBR	cbr.cmp	.app
Cellular Call	cellular_abstract.cmp	.app
FTP	ftp.cmp	.app
FTP Generic	ftpgeneric.cmp	.app
GSM Call	gsm.cmp	.app
HTTP	http.cmp	.app
HTTPD	httpd.cmp	.app
Lookup	lookup.cmp	.app
MCBR	mcbr.cmp	.app
Super Application	superapplication.cmp	.app
Super Application (Single Host)	superapplication_singlehost.cmp	.app
VoIP	voip.cmp	.app
TELNET	telnet.cmp	.app
Traffic Generator	trafficgen.cmp	.app
Traffic Generator (Single Host)	trafficgen_singlehost.cmp	.app
Traffic Trace	traffictrace.cmp	.app
Traffic Trace (Single Host)	traffictrace_singlehost.cmp	.app
UMTS Call	umtscall.cmp	.app
VBR	vbr.cmp	.app
VoIP	voip.cmp	.app
ZigbeeApp	zigbeeapp.cmp	.app

5.1.1.3 Shared Description Files

Shared description files contain descriptions of property editor segments shared by component files. Table 5-2 lists the different shared description files used by Architect and the configuration file generated by them. The second column of table also lists the `category` and `subcategories` defined in the file. The indentations in the table entry correspond to the levels of nesting in the file. For example, file application.prt has one `category`, `NODE-CONFIGURATION`, which has one `subcategory`,

APPLICATION, as its child. The other six subcategories are children of the subcategory APPLICATION. The shared description files are stored in EXATA_HOME/gui/settings/protocol_models.

TABLE 5-2. Shared Description Files

Shared Description File	Explanation	Configuration File Generated
antenna.prt	Properties of antenna models. category name="NODE CONFIGURATION subcategory name="Antenna Properties"	.antenna-models
application.prt	Structural information for all applications. category name="NODE CONFIGURATION subcategory name="APPLICATION" subcategory name="HTTP" subcategory name="DNS" subcategory name="VOIP" subcategory name="IPNE" subcategory name="RTP" subcategory name="MDP"	.app
application_layer.prt	Common properties of all applications. category name="NODE CONFIGURATION" subcategory name="Application Layer"	.config
arp.prt	Properties of the ARP model. category name="NODE CONFIGURATION" subcategory name="ARP"	.config
atminterfaces.prt	Detailed properties of ATM interfaces. category name="NODE CONFIGURATION" subcategory name="ATM INTERFACES" subcategory name="ATM Interface 0" subcategory name="ATM Layer2" subcategory name="Adaptation Protocols" subcategory name="ARP"	.config
battery_models.prt	Properties of battery models. category name="NODE CONFIGURATION" subcategory name="Battery Models"	.config
channel_properties.prt	Properties of radio channels including frequency and propagation characteristics (pathloss, fading, and shadowing). category name="NODE CONFIGURATION" subcategory name="Channel Properties"	.config
external_interfaces.prt	Properties of standard external interfaces, such as VR-Link and AGI. category name="NODE CONFIGURATION" subcategory name="External Interfaces" subcategory name="VR-Link Interface" subcategory name="HLA Interface" subcategory name="DIS Interface" subcategory name="AGI Interface" subcategory name="Socket Interface" subcategory name="Warm-up Phase"	.config

TABLE 5-2. Shared Description Files (Continued)

Shared Description File	Explanation	Configuration File Generated
interfaces_device_properties.prt	Properties of external interface devices. category name="NODE CONFIGURATION" subcategory name="INTERFACE DEVICE PROPERTIES”	.config
interface_faults.prt	Properties of interface faults. category name="NODE CONFIGURATION” subcategory name="INTERFACE FAULTS” subcategory name=”Interface Fault 0"	.config, .faults
interfaces.prt	Detailed properties of interfaces. category name=”NODE CONFIGURATION" subcategory name="INTERFACES" subcategory name="Interface 0" subcategory name="Physical Layer" subcategory name="MAC Layer" subcategory name="Internet Controller" subcategory name="Network Layer" subcategory name="Routing Protocol" subcategory name="Faults" subcategory name="File Statistics"	.config
mac_layer.prt	Properties of MAC layer protocols. category name="NODE CONFIGURATION" subcategory name="MAC Layer"	.config
mobility.prt	Mobility-related properties. category name="NODE CONFIGURATION" subcategory name="Mobility"	.config
mpls.prt	MPLS properties. category name="NODE CONFIGURATION" subcategory name="MPLS Specs"	.config
network_layer.prt	Properties of Network Layer protocols. category name="NODE CONFIGURATION" subcategory name="NETWORK LAYER" subcategory name="Schedulers and Queues" subcategory name="QoS Configuration" subcategory name="ARP" subcategory name="Fixed Communications"	.config
network_management.prt	Parameters for Network Management models. category name="NODE CONFIGURATION" subcategory name="Network Management"	.config
node_faults.prt	Properties of node faults. category name="NODE CONFIGURATION" subcategory name="NODE FAULTS” subcategory name="Node Fault 0"	.config, .faults
packet_tracing.prt	Properties for packet tracing. category name="NODE CONFIGURATION" subcategory name="PACKET TRACING"	.config

TABLE 5-2. Shared Description Files (Continued)

Shared Description File	Explanation	Configuration File Generated
phy_layer.prt	Properties of Physical Layer protocols. category name="NODE CONFIGURATION" subcategory name="Physical Layer"	.config
point_to_point_link.prt	Properties of point-to-point links. category name="COMPONENTS" subcategory name="LINKS" subcategory name="POINT TO POINT LINK PROPERTIES" subcategory name="Network Protocol" subcategory name="Fixed Communications" subcategory name="Routing Protocol" subcategory name="ARP" subcategory name="Faults" subcategory name="Background Traffic" subcategory name="Background Traffic 0" subcategory name="File Statistics"	.config, .bgtraffic, .faults
router_models.prt	Properties of router models. category name="NODE CONFIGURATION" subcategory name="ROUTER MODEL"	.config
routing_protocols.prt	Properties of routing protocols. category name="NODE CONFIGURATION" subcategory name="ROUTING PROTOCOL" subcategory name="BGP Configuration" subcategory name = “BGP Router-Id” subcategory name = “Networks Advertises” subcategory name = “Neighbors”	.config, .bgp
statistics.prt	Parameters for collecting statistics for .stat file. category name="NODE CONFIGURATION" subcategory name="STATISTICS"	.config
statistics_database.prt	Parameters for collecting statistics for statistics database. category name="NODE CONFIGURATION" subcategory name="Statistics Database"	.config
statistics_database_node_level.prt	Node-level parameters for collecting statistics for statistics database. category name="NODE CONFIGURATION" subcategory name="STATISTICS DATABASE NODE LEVEL"	.config
supplemental_file.prt	List of supplemental files used in the scenario. category name="NODE CONFIGURATION" subcategory name="Supplemental Files”	.config
terrain.prt	Terrain properties. category name="NODE CONFIGURATION" subcategory name="Terrain”	.config

TABLE 5-2. Shared Description Files (Continued)

Shared Description File	Explanation	Configuration File Generated
transport.prt	Properties of the Transport Layer. category name="NODE CONFIGURATION" subcategory name="TRANSPORT"	.config
user-behavior.prt	User behavior model properties. category name="NODE CONFIGURATION" subcategory name="USER BEHAVIOR"	.config

5.1.2 Elements of Settings Files

This section describes the elements used in the GUI settings files and their attributes.

5.1.2.1 The `category` Element

The `category` element is the top-level element in the all GUI settings files. A `category` element can have `subcategory` and `variable` elements as its children. In a component file, the `category` element represents the property editor. In a shared description file, the `category` element represents the group of shared segment descriptions. The attributes of the `category` element are listed in Table 5-3.

TABLE 5-3. Attributes of the `category` Element

Attribute Name	Attribute Values or Type	Description
`name` *Required*	String	Name of the `category`. For a component file, this is the name displayed in the associated property editor's title bar.
`addon` *Optional*	Comma-separated list of strings	Name(s) of the addon module(s) in which this `category` is available. At least one of the listed addon modules should be installed for this `category` to be available. **Note: For Scalable Network Technologies use only.**
`propertytype` *Required for component files* *Optional for shared description files*	String	Component identifier. The `propertytype` should be unique across all component files.
`displaynme` *Required for component files*	String	This attribute is used only in component files representing applications and specifies the name of the application that is displayed on the canvas.

TABLE 5-3. Attributes of the `category` Element

Attribute Name	Attribute Values or Type	Description
`singlehost` *Optional*	List: • `true` • `false`	This attribute is used only in component files representing applications and specifies whether the application is a single-host application. `true` : Application is a single-host application. `false` : Application is a client-server application.
`Loopback` *Optional*	List: • `true` • `false`	This attribute is used only in component files representing applications and specifies whether the application is loop-back enabled. `true` : Application is a loop-back enabled. `false` : Application is not loop-back enabled.

The following is an example of a `category` element representing the default device property editor:

```
<category name="Default Device Properties" icon="default.gif"
          propertytype="Device">
     <variable name="Node Name" key="HOSTNAME" type="Text"
               default="Host" help="" unique="true" />
     <variable name="2D Icon" key="GUI-NODE-2D-ICON" type="Icon"
               default="default.png" help="" invisible="ScenarioLevel"
               filetype="xpm,bmp,jpg,jpeg,png"/>
     <variable name="3D Icon" key="GUI-NODE-3D-ICON" type="File"
               default="default.3ds" help="" />
     <variable name="Partition" key="PARTITION" type="Integer"
                  default="0"
                  help="Parallel partition to which to assign node." />
    <subcategory name="Node Configuration" icon="nodeconfig.gif">
         <subcategory name="Mobility and Placement"
                      refpath="NODE CONFIGURATION+Mobility"
                      icon="protocol.gif" />
        <subcategory name="Network Layer"
                     refpath="NODE CONFIGURATION+Network Layer"
                     icon="protocol.gif" />
        <subcategory name="Routing Protocol"
                      refpath="NODE CONFIGURATION+ROUTING PROTOCOL"
                     icon="protocol.gif" />
         ...
   </subcategory>
   <subcategory name="Interfaces"
                refpath="NODE CONFIGURATION+INTERFACES"
                icon="interfaces.gif" />
</category>
```

5.1.2.2 The `subcategory` Element

A `subcategory` element can have `subcategory` and `variable` elements as its children. The `subcategory` element contains information for displaying a segment (tab or list item) in a property editor. The attributes of the `subcategory` element are listed in Table 5-4.

TABLE 5-4. Attributes of the `subcategory` Element

Attribute Name	Attribute Values or Type	Description
`name` *Required*	String	Specifies the name of the `subcategory`. When the `subcategory` is included in a component file (either directly or through a reference), this name is displayed as the name of a tab or list item of a property editor.
`addon` *Optional*	Comma-separated list of strings	Name(s) of the addon module(s) in which this `subcategory` is available. At least one of the listed addon modules should be installed for this `subcategory` to be available. **Note: For Scalable Network Technologies use only.**
`refpath` *Optional*	String	Reference path. This is a concatenation of `category` and `subcategory` names that identifies the location of a segment description in shared description files. See Section 5.1.3 for details. **Note: It is recommended that this attribute be used only only in component (.cmp) files an not in shared description (.prt) files.**
`help` *Optional*	String	Help text displayed when the mouse is placed over the parameter group associated with the subcategory in the property editor.
`invisible` *Optional*	Comma-separated list of strings	Categories in which this subcategory is not invisible. This attribute is used to specify the property editors in which a shared parameter group is invisible. The categories are specified by their `propertytype` attribute. **Note: Not recommended for users.**

The following are examples of a `subcategory` element:

```
<subcategory name="General">
   <variable name="Node Name" key="HOSTNAME" type="Text"
             default="Switch" help="" />
   <variable name="2D Icon" key="GUI-NODE-2D-ICON" type="File"
             default="switch.gif" help="" />
   <variable name="3D Icon" key="GUI-NODE-3D-ICON" type="File"
             default="switch.3ds" help="" />
</subcategory>
...
<subcategory name="Node Configurations" icon="nodeconfig.gif">
   <subcategory name="Mobility" refpath="NODE CONFIGURATION+MOBILITY"
                icon="protocol.gif" />
   ...
</subcategory>
```

5.1.2.3 The `variable` Element

A `variable` element can have `variable` and `option` elements as its children. The `variable` element corresponds to a parameter in a property editor. The attributes of the `variable` element are listed in Table 5-5.

TABLE 5-5. Attributes of the `variable` Element

Attribute Name	Attribute Values or Type	Description
`name` *Required*	String	Name of the `variable`. When the `variable` is included in a component file (either directly or through a reference), this name is displayed as a parameter name in a property editor.
`key` *Required*	String	Identifier (parameter name) printed to the configuration file, for example, `SEED` or `SIMULATION-TIME`.
`type` *Required*	List See Table 5-6.	Type of the `variable`. This attribute determines the type of the associated parameter and the specialized component used to accept the value of the parameter. For example, if `type` is `Selection`, then the parameter can take a value from a list, and a combo-box with possible values is displayed.
`default` *Required*	See Table 5-6.	Default value of the parameter represented by the `variable`. The default value depends upon the `type`. See Table 5-6.
`help` *Optional*	String	Help text that explains the purpose of the parameter associated with the `variable` and is typically displayed as a tool-tip when the mouse is placed over the parameter name in the property editor.
`min`, `max` *Optional*	Integer, Real, or IPv4 address (depending on the `type` attribute)	Minimum and maximum values for the parameter associated with the `variable`. These are used to specify the range of values for the parameter. These are used only if the `variable` is of type integer, fixed, or dotted decimal (see Table 5-6).

TABLE 5-5. Attributes of the `variable` Element (Continued)

Attribute Name	Attribute Values or Type	Description
`unit` *Optional*	String	Unit used for representing the values of the parameter associated with the `variable`, e.g., `unit = "bps"`
`requires` *Optional*	JavaScript Boolean Expression	Condition that should be satisfied to accept the value entered for the `variable`. This condition can be expressed as a JavaScript expression (see below.)
`disable` *Optional*	JavaScript Boolean Expression	Condition that should be satisfied for this `variable` to be read-only. This condition can be expressed as a JavaScript expression (see below.) **Note: Not recommended for users.**
`filetype` *Optional*	Comma-separated list of file extensions	Recommended file types. This is used when the `type` attribute is set to `File`.
`invisible` *Optional*	Comma-separated list of values of the `propertytype` attribute of `category` elements in component files	`propertytype` attribute of the `categories` in which this `variable` is not visible. This attribute is used to specify the property editors in which a shared parameter is invisible. **Note: Not recommended for users.**
`maxunit`, `minunit` *Optional*	Decimal abbreviation optionally followed by a unit (must be same as the `unit` attribute) Supported abbreviations (which are case-insensitive) in order from largest to smallest are: `E`, `P`, `T`, `G`, `M`, and `K` *Default abbreviation for `maxunit`*: `E` *Default abbreviation for `minunit`*: none (no abbreviation)	Maximum and minimum units. These attributes are used only if the `variable` is of type fixed multiplier. These attributes determine the range of units that are available for specifying the parameter's value. For example, if `maxunit = "Tbps"` and `minunit` is not specified, then the following units are available from the units combo-box: `Tbps`, `Gbps`, `Mbps`, `Kbps`, and `bps`. **Note**: A numerical value followed by a unit can also be specified for these attributes, e.g., `maxunit = "10 Gbps"`. In that case, the value denoted by this attribute is the maximum (minimum) value of the parameter associated with the variable.
`addon` *Optional*	Comma-separated list of strings	Name(s) of the addon module(s) in which this `variable` is available. At least one of the listed addon modules should be installed for this `variable` to be visible. **Note: For Scalable Network Technologies use only.**

`prepend`
Optional
	List:
	• `nn`
	• `id`
		Indication that the `key` is printed with a qualifier prepended to it.
		`nn` : Indicates that the network identifier is printed before the `key`
		`id` : Indicates that the node identifier is printed before the `key`

TABLE 5-5. Attributes of the `variable` Element (Continued)

Attribute Name	Attribute Values or Type	Description
`unique` *Optional*	List: • `true` • `false`	Indication whether this property can be modified in group editing mode or not. `true` : Indicates that the property can not be edited in group editing mode `false` : Indicates that the property can be edited in group editing mode
`keyvisible` *Optional*	List: • `true` • `false`	Indication whether the `key` associated with this `variable` will be printed in the scenario configuration (.config) file. `true` : The `key` will be printed in the scenario configuration file. `false` : The `key` will not be printed in the scenario configuration file. **Note: Not recommended for users.**
`optional` *Optional*	List: • `true` • `false`	Indicates whether the a value is required for the parameter represented by this `variable`. `true` : A value is not required for the parameter. `false` : A value is required for the parameter.
`visibilityrequires` *Optional*	JavaScript Boolean Expression	Condition that should be satisfied to make the `variable` visible in the property editor. This condition can be expressed as a JavaScript expression (see below.)
`interfaceindex` *Optional*	List: • `true` • `false`	Indicates whether the parameter, when configured at the interface level, is saved in the scenario configuration (.config) file with a Node ID and an instance corresponding to the interface, or with the interface address. `true` : The parameter is saved with the node ID and instance. `false` : The parameter is saved with the interface address. By default, interface-level parameters are saved with interface addresses.

TABLE 5-5. Attributes of the `variable` Element (Continued)

Attribute Name	Attribute Values or Type	Description
`help_ref` *Optional*	string	String used to associate the parameter name with the correct document in the model libraries (for use in context-sensitive help). This is useful when the same parameter name is used in more than one model. The format of the string is: `<Doc Title>:<Description>` where `<Doc Title>` : Title of the document in which the parameter is described. `<Description>` : Part or all of the text used to describe the parameter in the Description column in the parameter table. **Note: Not recommended for users.**
`spacesAllowed` *Optional*	List: • `true` • `false`	Indicates whether the value of the parameter associated with the `variable` can contain spaces (if the `variable` is of type text). `true` : Spaces are allowed in the `value`. `false` : Spaces are not allowed in the `value`. By default, spaces are allowed in the value.
`issatellitekey` *Optional*	List: • `true` • `false`	Indicates whether the `key` associated with the `variable` is a parameter of the satellite model. `true` : `key` is a parameter of the satellite model. `false` : `key` is not a parameter of the satellite model. By default, the `key` is not a parameter of the satellite model. **Note: For Scalable Network Technologies use only.**
`dontwritetoconfig` *Optional*	List: • `true` • `false`	Indicates whether the `key` and `value` associated with the `variable` should be printed in the scenario configuration (.config) file. `true` : `key` and `value` are not printed in the scenario configuration file. `false` : `key` and `value` are printed in the scenario configuration (.config) file By default, the `key` and `value` are printed in the scenario configuration file. **Note: Not recommended for users.**

TABLE 5-5. Attributes of the `variable` Element (Continued)

Attribute Name	Attribute Values or Type	Description
`refreshtable` *Optional*	List: • `true` • `false`	Indicates whether the property editor is automatically refreshed when a dependent parameter of the `variable` is changed. `true` : Property editor is refreshed automatically. `false` : Property editor is not refreshed automatically. By default, the property editor is not refreshed automatically.

JavaScript Expressions

JavaScript expressions can be used in XML files to represent boolean conditions. JavaScript expressions can be used as the value of the `requires`, `disable`, and `visibilityrequires` attributes of the `variable` element and of the `requires` and `visibilityrequires` attributes of the `option` element. In GUI settings files, JavaScript expressions must follow these rules:

- The `key` attribute of a `variable` element can be used in JavaScript expressions if it is enclosed in square brackets, '`[`' and '`]`' (except when used as an argument of macros `scriptInterface.anyEqualsTo` and `scriptInterface.allEqualsTo`, as described below). For example `AODV-HELLO-INTERVAL` can be used in a JavaScript expression as follows:

```
[AODV-HELLO-INTERVAL] > 3000000000
```

- The `value` attribute of an `option` elements can be used in JavaScript expressions if it is enclosed in single quotes. For example, `IPv6` and `DUAL-IP` can be used in a JavaScript expression as follows:

```
[NETWORK-PROTOCOL] == 'IPv6' || [NETWORK-PROTOCOL] == 'DUAL-IP'
```

- Macro `this.value` can be used in a JavaScript expression to refer to the value of the `variable` element in whose attributes the expression is used. For example:

```
this.value >= [DYMO-TTL-START]
```

- Time values used in JavaScript expressions must be in units of nanoseconds. For example, the following expression states the condition that the current value is greater than 2 seconds:

```
this.value > 2000000000
```

  Time unit abbreviations, such as `S`, `MS`, `H`, etc., should not be used in Java expressions. Expressions using these abbreviations will always evaluate to true.

- For a `variable` element of type `array`, use an index enclosed in parentheses, '`(`' and '`)`' after the `key` attribute to refer to a specific instance of the `variable` element. For example:

```
[PROPAGATION-MODEL(2)]=='FREE-SPACE'
```

- For a `variable` element of type `array`, use the macro `scriptInterface.anyEqualsTo` if *any* instance of the `variable` element can have a specified value. In this case, the `key` attribute should be

enclosed in single quotes. For example, the following expression is true if any instance of PROPAGATION-MODEL is set to FREE-SPACE.

```
scriptInterface.anyEqualsTo('PROPAGATION-MODEL','FREE-SPACE')
```

- For a variable element of type array, use the macro scriptInterface.allEqualsTo if *all* instances of the variable element should have a specified value. In this case, the key attribute should be enclosed in single quotes. For example, the following expression is true if all instances of PROPAGATION-MODEL is set to FREE-SPACE.

```
scriptInterface.allEqualsTo('PROPAGATION-MODEL','FREE-SPACE')
```

- The macro respective can be used as the dynamic index to bind the instances of two arrays if the variable elements representing the two arrays are descendents of the same variable element of type array. Consider the following example:

```
<variable name="Number of Queues"
          key="NUM-PRIORITIES" type="Array" default="3" min="1">
  <variable name="Queue Size" key="QUEUE-SIZE" type="Integer"
            default="150000" />
  <variable name="Queue Type" key="QUEUE-TYPE" type="Selection"
            default="FIFO">
     <option value="FIFO" name="FIFO"/>
     <option value="RED" name="RED"/>
     <option value="WRED" name="WRED">
       <variable name="Minimum Threshold" key="MIN-THRESHOLD"
                 type="Integer" default="10"
                visibilityrequires="[QUEUE-TYPE(respective)]=='WRED'"/>
     </option>
    </variable>
  </variable>
```

In this example, Queue Size and Queue Type are children of Number of Queues, which is a variable of type array. Queue Type has several options, one of which, WRED, has a child, Minimum Threshold. Minimum Threshold is visible only if the Java expression [QUEUE-TYPE(respective)]=='WRED' is true. This expression evaluates to true only if the corresponding instance of QUEUE-TYPE is set to WRED, for example, Minimum Threshold[2] will be visible only if Queue Type[2] is set to WRED. (Note that both Minimum Threshold and Queue Type are descendents of same variable of type array, Number of Queues.)

Table 5-6 lists the acceptable values for the type attribute of the variable element and the corresponding values of the default attribute. Note that not all values are available in all the XML files.

TABLE 5-6. Possible Values of type and Corresponding default Attributes of the variable Element

Value of the type Attribute	Description	Valid Values for the default Attribute
Integer	For integer inputs	Any valid integer value, e.g. "1", "199", etc.
Fixed	For floating point inputs	Any valid float value, e.g. "1000.50"
Checkbox	For YES/NO inputs via a combo box	"NO" or "YES"

TABLE 5-6. Possible Values of `type` and Corresponding `default` Attributes of the `variable` Element (Continued)

Value of the `type` Attribute	Description	Valid Values for the `default` Attribute
`File`	For selecting a file from the file system	One of the following: • `"[Required]"`: Indicates that the file parameter is required • `"[Optional]"`: Indicates that the file parameter is not required • A valid file path, e.g., `"./somefile.txt"`
`Time`	For time inputs	A time value in EXata time format, e.g., `"2S"`, `"500MS"`, or "5H".
`Mask`	For inputs that are strings of '0' and '1'	A string of 0's or 1's, e.g., `"01"`.
`Coordinates`	For accepting a set of two double values	Space-separated integer or float values, e.g., `"100.0 200.0"`.
`Fixed multiplier`	For accepting unit based input, such as 100 MHz, 2 Gbps, etc. If this type is used, the following attributes of the `variable` also need to be specified: `unit`, `minunit`, and `maxunit`.	Numerical value and the unit, separated by a space, e.g., `"10 mps"`.
`Text`	For accepting strings	Any string.
`Selection`	For accepting a value from a list. A combo-box with a list of items to be selected is displayed. Which items are displayed in the combo-box is determined by the `option` elements that are children of the `variable`.	The `value` of any of the `option` elements that are children of the `variable`. See Section 5.1.2.4.
`Tickbox`	For YES/NO inputs via a check-box	`"NO"` or `"YES"`
`Dotted decimal`	For accepting inputs in the format of an IPv4 address.	Any valid IPv4 address, e.g., `"192.0.1.1"`, `"128.1.234.1"`, etc.
`SlotFile`	For launching the Slot File Editor. A button is displayed to launch the Slot File Editor. The slot file name selected in the Slot File Editor is also displayed. **Note: Not recommended for users.**	`"Optional"`
`Array`	For accepting array variables. An `Array` variable specifies the number of child `variables`, e.g., the number of priority queues, the number of propagation channels, etc. **Note: Not recommended for users.**	A positive integer value.
`Icon`	For accepting icon file names.	Any image file.

TABLE 5-6. Possible Values of `type` and Corresponding `default` Attributes of the `variable` Element (Continued)

Value of the `type` Attribute	Description	Valid Values for the `default` Attribute
`NodeList`	For accepting list of node IDs (separated by spaces). **Note: Not recommended for users.**	A list of node IDs.
`SelectionDynamic`	For accepting a value from a list whose members are available only at run time. A combo-box with a list of items (node ID and its interface addresses) to be selected is displayed. Which items are displayed in the combo-box is determined at run time and depends on the component type. **Note: Not recommended for users.**	There is no default value for this type.
`NetworkCheckBox`	For accepting a value from a list of advertised networks for BGP. A combo-box with a list of networks is displayed. Which networks are displayed in the combo-box is determined at run time. **Note: Not recommended for users.**	There is no default value for this type.

Only a `variable` element of type `Array` can have `variable` elements as its children. If a `variable` is a child of a `variable` of type `Array`, the child variable corresponds to an indexed parameter. For example, a `variable` of type `Array` representing the parameter `IP-QUEUE-NUM-PRIORITIES` has other `variables` as its children which correspond to the parameters `IP-QUEUE-TYPE`, `QUEUE-WEIGHT`, etc.

Only a `variable` element of type `Selection` can have `option` elements as its children. Each `option` child of a `variable` corresponds to one of the possible values of the parameter represented by the `variable` element. For example, a `variable` of type `Selection` representing the parameter `ROUTING-PROTOCOL` has several `options` as its children, one for each routing protocol (AODV, LAR, DSR, etc.).

The following is an example of a variable element:

```
<variable name="Directional Antenna Mode"
          key="MAC-DOT11-DIRECTIONAL-ANTENNA-MODE" type="Selection"
          default="NO">
   <option value="NO" name="No" />
   <option value="YES" name="Yes">
       <variable name="Direction Cache Expiration Time"
                 key="MAC-DOT11-DIRECTION-CACHE-EXPIRATION-TIME"
                 type="Time" default="2S" />
       <variable name="NAV Delta Angle"
                 key="MAC-DOT11-DIRECTIONAL-NAV-AOA-DELTA-ANGLE"
                 type="Fixed" default="37.0" />
       <variable name="Short Packet Limit"
                key="MAC-DOT11-DIRECTIONAL-SHORT-PACKET-TRANSMIT-LIMIT"
                 type="Integer" default="8" />
   </option>
</variable>
```

5.1.2.4 The option Element

An option element can have variable elements as its children. The attributes of the option element are listed in Table 5-7.

TABLE 5-7. Attributes of the option Element

Attribute	Required or Optional	Description of the Attribute
name *Required*	String	Name of the option. No two options that are children of the same parent variable can have the same name. When the option is included in a component file (either directly or through a reference), this name is displayed in a combo-box as one of the values of the parent variable (parameter).
value *Required*	String	String that is written in the EXata configuration file as the value of the parent variable (parameter). No two options that are children of the same parent variable can have the same value.
addon *Optional*	Comma-separated list of strings	Name(s) of the addon module(s) in which this option is available. At least one of the listed addon modules should be installed for this option to be visible. **Note: For Scalable Network Technologies use only.**
requires *Optional*	JavaScript Boolean Expression	Condition that should be satisfied to accept the value entered for the option. This condition can be expressed as a JavaScript expression (see Section 5.1.2.3.)
visibilityrequires *Optional*	JavaScript Boolean Expression	Condition that should be satisfied to make the option visible in the property editor. This condition can be expressed as a JavaScript expression (see Section 5.1.2.3.)

The following is an example of an `option` element:

```
<option value="RIP" name="RIP">
     <variable key="RIP-VERSION" type="Selection"
               name="Version" default="2" optional="true" >
        <option value="1" name="1" />
        <option value="2" name="2" >
            <variable key="RIP-COMPATIBILITY"
                      type="Selection"
                      name="RIP Compatibility type"
                      default="RIPv2-ONLY" optional="true" >
                  <option value="RIPv2-ONLY" name="RIPv2 Only" />
                  <option value="RIPv1-ONLY" name="RIPv1 Only" />
                  <option value="RIPv1-COMPATIBLE"
                          name="RIPv1-Compatible" />
             </variable>
         </option>
     </variable>
    <variable name="Border Router" key="RIP-BORDER-ROUTER"
              type="Selection" default="NO" optional="true" >
         <option value="NO" name="No" />
         <option value="YES" name="Yes" >
              <variable name="Auto Summary" key="RIP-AUTO-SUMMARY"
                        type="Selection" default="NO" optional="true" >
                  <option value="NO" name="No" />
                  <option value="YES" name="Yes" />
              </variable>
         </option>
  </variable>
    ...
</option>
```

5.1.3 Using Shared Descriptions

As described in Section 5.1.1, a component file describes the structure of a property editor. A property editor is composed of one or more segments (tabs or list items). For each segment of the property editor, the component file either contains its detailed description or refers to a segment description in a shared description file.

This sharing of descriptions can only be done at the segment level. Since segments are represented by `subcategory` elements, `subcategories` in component files can refer to `subcategories` in shared description files. This is done by means of the `refpath` attribute of the subcategory element (see Section 5.1.2.2). The `refpath` attribute takes as its value a reference path. A reference path is a path to a subcategory definition derived by concatenating, in order, the root `category` name and all `subcategory` names along the path. All valid reference paths to `subcategories` in shared description files can be obtained by concatenating `category` and `subcategory` names in Table 5-2 such that a `subcategory` in the concatenated string is a child of the preceding `category` or `subcategory`. The '+" operator is used for concatenation.

Following are examples of valid reference paths derived from Table 5-2:

```
NODE CONFIGUARTION+INTERFACES FAULTS
NODE CONFIGURATION+NETWORK LAYER+QoS
COMPONENTS+LINKS+Background Traffic
```

Wherever a component file uses a reference in a subcategory description, the corresponding description from the shared description file is used to display the property editor segment. As an example consider the property editors for the default node (see Figure 5-1) and the wireless subnet (see Figure 5-2). Both property editors share the segment Routing Protocol, which appears as a list item in the default node property editor and as a tab in the wireless subnet property editor. The component file for the default node, defaultnode.cmp, is shown in Figure 5-3.

Note **It is recommended that shared description (.prt) files not contain references to other shared description files. Only component files should use references to shared description files.**

```
...
<category name="Default Device Properties" icon="default.gif"
          propertytype="Device">
     <variable name="Node Name" key="HOSTNAME" type="Text" default="Host"
               help="" unique="true" />
     ...
     <subcategory name="Node Configuration" icon="nodeconfig.gif">
        <subcategory name="Mobility and Placement"
                     refpath="NODE CONFIGURATION+MOBILITY"
                     icon="protocol.gif" />
        ...
        <subcategory name="Routing Protocol"
                     refpath="NODE CONFIGURATION+ROUTING PROTOCOL"
                     icon="protocol.gif" />
        ...
        <subcategory name="Faults"
                     refpath="NODE CONFIGURATION+NODE FAULTS"
                     icon="faults.gif" help="Specify card fault"/>
     </subcategory>
     <subcategory name="Interfaces"
                  refpath="NODE CONFIGURATION+INTERFACES"
                  icon="interfaces.gif" />
</category>
...
```

FIGURE 5-3. Component File for Default Node's Property Editor

The component file for the wireless subnet property editor is shown in Figure 5-4.

```
<category name="Wireless Subnet Properties" icon=""
          propertytype="WirelessSubnet">
     <variable name="2D Icon" key="GUI-NODE-2D-ICON" type="Icon"
               default="wireless.png" help="" invisible="ScenarioLevel"
               filetype="xpm,bmp,jpg,jpeg,png"/>
     ...
     <subcategory name="Physical Layer"
                  refpath="NODE CONFIGURATION+PHYSICAL LAYER"
                  icon="protocol.gif" />
     <subcategory name="MAC Layer"
                  refpath="NODE CONFIGURATION+MAC Layer"
                  icon="protocol.gif" />
     ...
     <subcategory name="Routing Protocol"
                  refpath="NODE CONFIGURATION+ROUTING PROTOCOL"
                  icon="protocol.gif" />
     ...
</category>
...
```

FIGURE 5-4. Component File for Wireless Subnet's Property Editor

Component files for both the default node and the wireless subnet use the reference path "`NODE CONFIGURATION+ROUTING PROTOCOL`" to use the description of the Routing Protocol segment in the shared description file network_type.prt, which is shown in Figure 5-5.

Note that component files for both the default node and the wireless subnet refer to the same description of the Routing Protocol segment, but the Routing Protocol segment appears at different places in the two property editors (see Figure 5-1 and Figure 5-2). This is because the default node component file refers to the Routing Protocol segment from the third level (`category -> subcategory -> subcategory`) while the wireless subnet component file refers to it from the second level (`category -> subcategory`).

```
<category name="NODE CONFIGURATION">
        <subcategory name="ROUTING PROTOCOL"
                     class="interface,device,atmdevice">
             <variable name="Routing Protocol IPv4"
                       key="ROUTING-PROTOCOL"
                       type="Selection" default="BELLMANFORD"
                       visibilityrequires="[NETWORK-PROTOCOL] != 'IPv6'">
              ...
            </variable>
              ...
            <variable name="Ensble IP Forwarding" key="IP-FORWARDING"
                      type="Checkbox" default="YES"
                      invisible="interface" optional="true"
                      help="Determines whether or not node(s) will forward
                            packets"/>
            ...
        </subcategory>
</category>
...
```

FIGURE 5-5. Shared Description of Routing Protocol Segment

5.1.4 Integrating New Models into Architect

In EXata, a new protocol can be developed manually using the procedures described in Chapter 4 and can be integrated into EXata Architect. This section describes how to modify the GUI settings files to integrate protocols into Architect.

Integrating an application protocol requires the creation of a new component file for the property editor of the application and possibly modifying one or more shared description file. This process is described in Section 5.1.4.2.

Integrating protocols other than application protocols requires modifying shared description files only. This process is described in Section 5.1.4.1.

Note **When modifying existing GUI settings files, do not delete or move any elements, as this will interfere with the proper working of existing property editors. Only add new elements needed to integrate your protocol, at the appropriate places, taking care to preserve the structure of the XML files and to maintain the rules for each element.**

Note **Changes made to the GUI settings files take effect only after the GUI is restarted.**

5.1.4.1 Integrating a New Protocol

To integrate a new protocol that is not an application protocol into EXata GUI, one of the shared description files described in Section 5.1.1 need to be modified. Identify a protocol from the EXata library that is most similar to the new protocol. The `subcategory` names listed in Table 5-2 indicate where each type of protocol is described in the shared description files. Find the segment of the shared description file pertaining to the existing protocol, and use that as a template to add code in the shared description file to incorporate the new protocol.

We illustrate the steps for integrating a protocol into EXata GUI by taking a new routing protocol as an example. Routing protocols are described in the `subcategory` identified by the path `NODE CONFIGURATION+ROUTING PROTOCOL` in the file routing_protocols.prt.

Figure 5-6 shows a code segment from file routing_protocols.prt that specifies different routing protocols. Each supported routing protocol appears as an `option` under the `variable` "`Routing Protocol IPv4`" or under the `variable` "`Routing Protocol IPv6`". If the routing protocol has any configurable parameters, they appear as `variable` elements under the `option` corresponding to the routing protocol. To add a new routing protocol that can be used only for IPv4 networks, add a new `option` under the `variable` "`Routing Protocol IPv4`", similar to the existing `options`. To add a new routing protocol that can be used only for IPv6 networks, add a new `option` under the `variable` "`Routing Protocol IPv6`". To add a new routing protocol that can be used for both IPv4 and IPv6 networks, add a new `option` at both places.

```
...
<subcategory name="ROUTING PROTOCOL">
     <variable name="Routing Protocol IPv4" key="ROUTING-PROTOCOL"
                    type="Selection" default="BELLMANFORD"
                    visibilityrequires="[NETWORK-PROTOCOL] != 'IPv6'">
        ...
        <option value="AODV" name="AODV" addon="wireless">
             <variable name="Network Diameter (hops)"
                       key="AODV-NET-DIAMETER" type="Integer" default="35"
                       help="The maximum possible number of hops between two
                             nodes in the network" />
             ...
        </option>
        <option value="DYMO" name="DYMO" addon="wireless">
                <variable name="Enable Processing Hello"
                          key="DYMO-PROCESS-HELLO" type="Selection"
                          default="NO" help="If the value is set to ...">
                    <option value="NO" name="No" />
                    <option value="YES" name="Yes">
                         ...
                    </option>
                </variable>
                ...
        </option>
        ...
        <!-- INSERT OPTION FOR NEW PROTOCOL FOR IPv4 or DUAL-IP HERE  -->
        ...
    </variable>
    <variable name="Routing Protocol IPv6" key="ROUTING-PROTOCOL-IPv6"
              type="Selection" default="RIPng"
              visibilityrequires="[NETWORK-PROTOCOL] == 'IPv6' ||
                                  [NETWORK-PROTOCOL] == 'DUAL_IP'">
       <option value="OSPFv3" name="OSPFv3" addon="multimedia_enterprise">
           <variable name="Define Area" key="OSPFv3-DEFINE-AREA"
                     type="Selection" default="NO"/>
           ...
       </option>
       ...
       <!-- INSERT OPTION FOR NEW PROTOCOL FOR IPV6 or DUAL-IP HERE  -->
       ...
       <option value="AODV" name="AODV" addon="wireless">
           ...
    </variable>
</subcategory>
...
```

FIGURE 5-6. Integrating a Routing Protocol

5.1.4.2 Integrating a New Traffic Generator

Integrating a new application protocol (traffic generator) into the EXata GUI involves the following steps:

- Creating a new component (.cmp) file in the folder EXATA_HOME/gui/settings/components
- Modifying the file Standard.xml in EXATA_HOME/gui/settings/Toolsets to display a button corresponding to the new application in the Standard Toolset of Architect.

Creating a Component File

To create a component file for a new application, identify a protocol from the EXata library that is most similar to the new protocol and use that protocol's component file as a template. Table 5-2 lists the available applications and their component files.

When creating a new component file, follow the rules for the file structure and elements described in Section 5.1.1 and Section 5.1.2. The component file should have one `category`. Each parameter of the protocol should be represented by one `variable`. Use `subcategories` to group parameters into tabs and list items, if desired.

Note: The `propertytype` of the new component should be assigned a value not used by any other component.

As an example, Figure 5-7 shows the component file for the CBR application, cbr.cmp, and Figure 5-8 shows the corresponding property editor. Each `variable` in the component file represents a CBR parameter. Each `option` represents a possible value for a parameter of enumeration type. Note that all top-level `variables` (direct children of the `category` element) are grouped under a default tab called "General".

```
...
<category name="CBR Properties" singlehost="false" loopback="enabled"
          propertytype="CBR">
   <variable name="Source" key="SOURCE" type="SelectionDynamic"
             keyvisible="false" optional="false"/>
   <variable name="Destination" key="DESTINATION" type="SelectionDynamic"
             keyvisible="false" optional="false"/>
   <variable name="Items To Send" key="ITEM-TO-SEND" type="Integer"
             default="100" min="0" keyvisible="false"
             help="Number of items to send" optional="false"/>
   <variable key="ITEM-SIZE" type="Integer" name="Item Size (bytes)"
             default="512" min="24" max="65023" keyvisible="false"
             help="Item size in bytes" optional="false"/>
   <variable name="Interval" key="INTERVAL" type="Time" default="1S"
             keyvisible="false" optional="false"/>
   <variable name="Start Time" key="START-TIME" type="Time" default="1S"
             keyvisible="false" optional="false"/>
   <variable name="End Time" key="END-TIME" type="Time" default="25S"
             keyvisible="false" optional="false"/>
   <variable name="Priority" key="PRIORITY" type="Selection"
             default="PRECEDENCE" keyvisible="false">
      <option value="TOS" name="TOS"
              help="value (0-255) of the TOS bits in the IP header">
         <variable name="TOS Value" key="TOS-BITS" type="Integer" default="0"
                   min="0" max="255" keyvisible="false" optional="false"/>
      </option>
      <option value="DSCP" name="DSCP"
              help="value (0-63) of the DSCP bits in the IP header">
        <variable name="DSCP Value" key="DSCP-BITS" type="Integer" default="0"
                   min="0" max="63" keyvisible="false" optional="false"/>
      </option>
      <option value="PRECEDENCE" name="Precedence"
              help="value (0-7) of the Precedence bits in the IP header">
        <variable name="Precedence Value" key="PRECEDENCE-BITS" type="Integer"
                   default="0" min="0" max="7" keyvisible="false"
                   optional="false"/>
      </option>
   </variable>
   <variable name="Enable Rsvp-Te" key="ENABLE-RSVP-TE" type="Selection"
             default="N/A" keyvisible="false" optional="true">
      <option value="N/A" name="No" />
      <option value="RSVP-TE" name="Yes" />
   </variable>
</category>
...
```

FIGURE 5-7. Component File for CBR Application

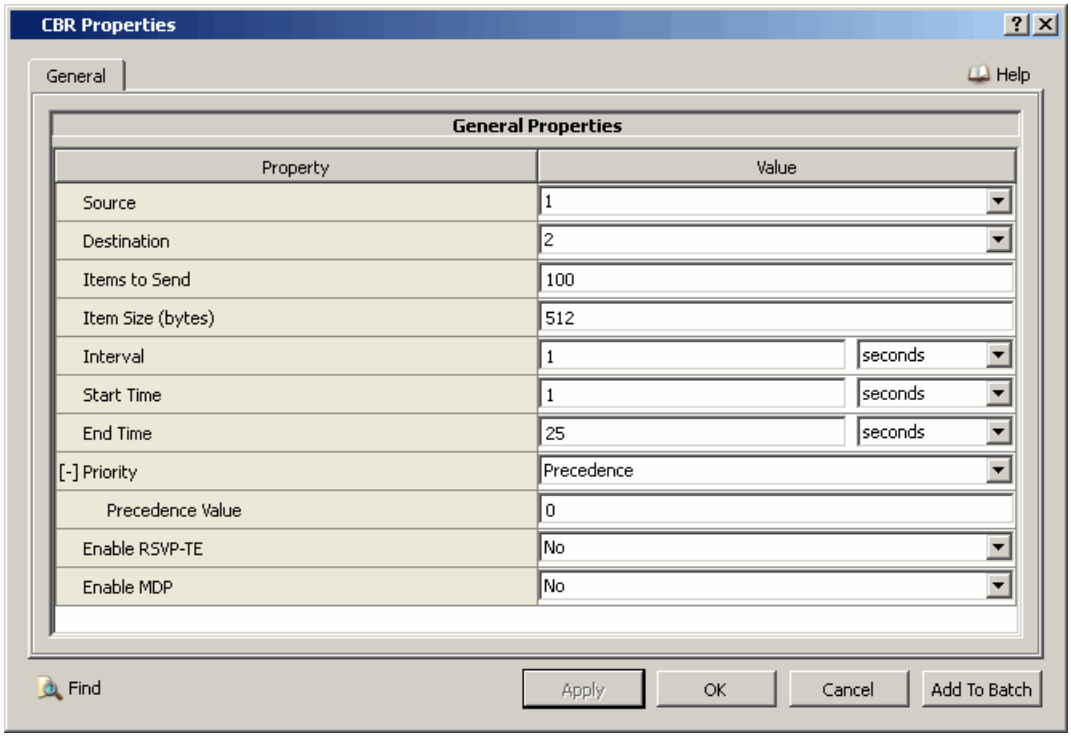


FIGURE 5-8. Property Editor for CBR Application

Adding a Button to the ToolSet

A button is added to the Standard Toolset by modifying the file Standard.xml. If the GUI has never been started, then you must modify the file Standard.xml in EXATA_HOME/gui/settings/toolsets. If the GUI has been started before, then you must modify the file Standard.xml in the following folder:

- In Windows 7 systems, modify the file Standard.xml in C:\Users\<username>\.exataUserDir\exata_5_1\Toolsets.
- In Linux systems, modify the file Standard.xml in ~/.exataUserDir/exata_5_1/Toolsets.

File Standard.xml is shown in Figure 5-9.

To add a button to the Toolset for the new application, add a `subcategory` under the `category` "Applications" with the following attributes:

- `name`: Name of the application.
- `icon`: Name of the image file for the button. This image file should be placed in the folder EXATA_HOME/gui/icons/3DVisualizer/icons. The image file should be in PNG format.
- `tooltip`: Text that is displayed when the mouse is placed over the button for the application.
- `type`: This should be "App" for all applications.
- `propertytype`: This should be the same as the `propertytype` of the `category` in the component file for the application.

```
...
<category name="Devices">
     <subcategory name="Default" icon="default.png" tooltip="Default"
                  type="Default" propertytype="Device" />
        <subcategory name="Switch" icon="switch.png" tooltip="Switch"
                     type="Switch" propertytype="Switch" />
        <subcategory name="ATM" icon="atm.png" tooltip="ATM" type="ATM"
                     propertytype="ATM" />
</category>

<category name="Applications">
    <subcategory name="CBR" icon="cbr.png" tooltip="CBR" type="App"
                 propertytype="CBR" />
    <subcategory name="Ftp" icon="ftp.png" tooltip="Ftp" type="App"
                 propertytype="FTP" />
    <subcategory name="Telnet" icon="telnet.png" tooltip="Telnet" type="App"
                 propertytype="TELNET" />
    <subcategory name="Ftp-Generic" icon="ftp_gen.png" tooltip="Ftp/Generic"
                 type="App" propertytype="FTP/GENERIC" />
    <subcategory name="Lookup" icon="lookup.png" tooltip="Lookup" type="App"
                 propertytype="LOOKUP" />
    ...
    <!-- INSERT ENTRY FOR NEW APPLICATION HERE  -->
    ...
    <subcategory name="VBR" icon="vbr.png" tooltip="VBR" type="App"
                 propertytype="VBR" />
</category>

<category name="Links">
...
</category>
...
```

FIGURE 5-9. File Standard.xml

5.2 Customizing Visualize Mode of EXata Architect

This section describes the customization features available for the Visualize mode of EXata Architect. Section 5.2.3 describes the communication between EXata Simulator and EXata Architect. Section 5.2.2 describes the API functions that can be used to add customized animation to a protocol. This information can be used to interface EXata Simulator with any other GUI module. Section 5.2.3 describes how to display dynamic statistics as they change during model execution.

5.2.1 Communication between EXata Simulator and EXata Architect

EXata Architect provides a graphical interface for animating experiments, and provides limited opportunity to interact with the running simulation. This section describes how the runtime interaction between the EXata Architect and Simulator is set up. A programmer can use the information presented in this section to interface EXata Simulator with a GUI module of their choice.

5.2.1.1 Initializing EXata

When you press the **Run** button in EXata Architect, the following steps are performed to initialize EXata for a simulation:

1. The internal representation of the scenario is written into the plain text format recognized by Simulator. These are the EXata configuration files, .config, .app, etc.
2. These files are placed a directory having the same name as the scenario and placed at the same level as the scenario file.
3. A socket is opened on the first available port ≥ 4000.
4. An external process is started with the following command:

```
exata <input-file> -interactive <local-host> <port#>
```

5. A connection on the socket is waited on.

EXata Simulator does the following:

1. Opens a socket connection to the specified host and port.

   Note **If EXata is run on a distributed architecture, each partition opens a socket connection to the specified host and port.**

2. Checks out a license.
3. Reads the input file.
4. Sends scenario initialization information to Architect by calling GUI_Initialize.
5. Creates and initializes nodes, networks and interfaces.
   - Calls GUI_InitNode for each node.
   - Calls GUI_CreateSubnet for each subnet.
   - Calls GUI_CreateHierarchy for hierarchical networks.
   - Calls GUI_InitialInterfaces to configure IP addresses for all the nodes.

     Note **This in turn may result in calls to other GUI functions.**

   - Calls GUI_InitWirelessInterface to set up the wireless properties (power, sensitivity, range etc.) for a node's wireless interface.
   - Calls GUI_CreateWeatherPattern for creating weather patterns.
6. Processes instructions sent from Architect.
7. Upon receiving a **Step** command from Architect, starts processing events.

Steps 2-3 are essentially identical to running Simulator from the command line. Steps 1 and 4 establish the communication channel between EXata Simulator and Architect. Step 5 is basically identical to the non-animated mode of execution, with occasional information sent to Architect to configure the network for viewing. Steps 6 and 7 are described more fully in Section 5.2.1.2.

5.2.1.2 Runtime Interaction

The interaction between Architect and Simulator is very simple. Architect sends commands to Simulator over the socket. At user defined intervals, Simulator reads the commands and adjusts its behavior accordingly. Simulator also sends event-related animation and statistics information to Architect as the events occur. Most of the information sent to Architect can be enabled or disabled through Architect controls. Table 5-8 lists the commands sent from Architect to Simulator.

TABLE 5-8. Commands Sent from Architect to Simulator

Command	Parameters	Description
GUI_STEP		This command tells Simulator to begin processing events for a period of simulation time specified by the GUI_SET_COMM_INTERVAL command. At the end of this period, Simulator waits for the next command from Architect.
GUI_SET_COMM_INTERVAL	Time (interval)	Tells Simulator how far to advance in simulation time before checking for more commands from Architect.
GUI_ENABLE_LAYER	Integer (layer #)	All animation commands from the specified protocol layer are sent over the socket to Architect.
GUI_DISABLE_LAYER	Integer (layer #)	All animation commands from the specified layer are dropped without being sent to Architect. Provides an easy way to limit the amount of information being passed over the socket. However, the animation functions are still called from the protocol code. The filtering is done in the GUI interface code.
GUI_ENABLE_NODE	Integer (node ID)	Enables animation for a particular node.
GUI_DISABLE_NODE	Integer (node ID)	Disables animation for a particular node.
GUI_SET_STAT_INTERVAL	Time (interval)	Sets the interval (in simulation time) at which the user can query Simulator for the latest statistics.
GUI_ENABLE_METRIC	Integer (metric ID) Integer (node ID) Integer (layer)	Enables transmission of a specific metric across the socket.
GUI_DISABLE_METRIC	Integer (metric ID) Integer (node ID) Integer (layer)	Disables reporting of a specific metric.
GUI_STOP		Terminates the simulation experiment.
GUI_PAUSE		Pauses the simulation experiment.
GUI_RESUME		Resumes the simulation experiment.
GUI_DYNAMIC_ReadAsString	String (object ID)	Instructs the simulator to read the value of the specified object. (The simulator will send the read value to the GUI.)
GUI_DYNAMIC_WriteAsString	String (object ID) String (new-value)	Instructs the simulator to update the value of the specified object to the specified new value.
GUI_DYNAMIC_ExecuteAsString	String (object ID) String (command)	Instructs the simulator to execute the specifed command on the specified object.

TABLE 5-8. Commands Sent from Architect to Simulator (Continued)

Command	Parameters	Description
GUI_DYNAMIC_Listen	String (object ID)	Indicates to the simulator that the GUI has started listening to the specified object in the dynamic hierarchy. (The simulator will start sending any updates to the object to the GUI.)
GUI_DYNAMIC_Unlisten	String (object ID)	Indicates to the simulator that the GUI has stopped listening to the specified object in the dynamic hierarchy. (The simulator will stop sending any updates to the object to the GUI.)
GUI_ENABLE_EVENT	Integer (event ID)	Enables the transmission of the specified event. The event ID is the number corresponding to the GUI event in the enumeration GuiEvents in file gui.h.
GUI_DISABLE_EVENT	Integer (event ID)	Disables the transmission of the specified event. The event ID is the number corresponding to the GUI event in the enumeration GuiEvents in file gui.h.
GUI_USER_DEFINED	String (HITL command)	Sends the specified HITL command to the simulator.
GUI_SET_ANIMATION_FILTER_FREQUENCY	Time (interval)	Used to filter similar animations if they occur with the specified frequency. **(??)**

The commands are sent across the socket as character strings: the command number followed by the parameters, if any. The command number is specified in the enumeration GuiCommands in the file gui.h. For example, the command to set the communication interval to 100 milliseconds is:

```
1 100000000
```

where: 1 is the command number for GUI_SET_COMM_INTERVAL, and
100000000 is the time representation of 100 milliseconds. (100MS is also an acceptable representation for 100 milliseconds.)

Table 5-9 lists the replies sent by Simulator to Architect.

TABLE 5-9. Replies Sent from Simulator to Architect

Reply	Parameters	Description
GUI_STEPPED	Time (the simulation time just reached)	Reports to Architect that Simulator has reached the end of the period specified by the last STEP command.
GUI_ANIMATION_COMMAND	Varies	Sends an animation command. The first parameter is the number corresponding to the GUI event in the enumeration GuiEvents in file gui.h. The remaining parameters depend upon the animation command.
GUI_ASSERTION	String (the error message)	Indicates an assertion failure in Simulator code.
GUI_ERROR	String (the error message)	Indicates an error was detected in Simulator code.
GUI_WARNING	String (the error message)	Indicates a warning from Simulator.
GUI_SET_EFFECT	5 integers (event, layer, type, effect, color)	Assigns a visual effect to a particular animation event.
GUI_STATISTICS_COMMAND	Varies	Sends a statistics command. The first parameter is the number corresponding to the GUI statistics event in the enumeration GuiStatisticsEvents in gui.h. The remaining parameters depend upon the statistics command.
GUI_FINISHED		Indicates the end of the simulation.
GUI_FINALIZATION_COMMAND	String (node ID)	Indicates the finalization of the specified node.
GUI_DYNAMIC_COMMAND	Varies	Creates or modifes an object in the dynamic hierarchy. If the object does not already exist, it is created with the specified value. if the object already exists, its value is modifed to the speciifed value. The first parameter is the number corresponding to the dynamic command in the enumeration GuiEvents in file gui.h. The remaining parameters depend upon the dynamiccommand.

The replies are sent across the socket as character strings (similar to sending commands): the reply number followed by the parameters, if any. The reply number is specified in the enumeration GuiReplies in the file gui.h

5.2.1.3 Finalization

When the simulation is run with Architect, simulation execution can end in one of three ways:

- The simulation runs to completion and terminates normally. A GUI_FINISHED reply is sent from Simulator to Architect.
- The simulation terminates with an error. A GUI_ERROR or GUI_ASSERTION reply is sent to Architect, Architect performs some clean up tasks and sends a GUI_STOP command to Simulator, and Simulator shuts down by sending a GUI_FINISHED reply back to Architect.

- The user terminates the execution by pressing Architect's stop button. Architect performs some clean up tasks and sends a GUI_STOP command to Simulator. After receiving GUI_STOP command, Simulator finalizes each node and sends a GUI_FINALIZATION_COMMAND command for each node to Architect. If Architect does not receive a GUI_FINISHED command for more than 15 seconds then it displays an Abort dialog. This dialog shows the current status of simulator and remains active until Architect receives a GUI_FINISHED command or the user selects the Abort button displayed in Abort dialog.

5.2.2 Adding Customized Animation to a Protocol

EXata provides a rich API for adding animation to a protocol. These API functions are defined in EXATA_HOME/include/gui.h and in *API Reference Guide*.

When running EXata from the command line, the following command enables animation output and the animation commands are dumped to standard output:

```
exata <input-file> -animate
```

Animation can also be enabled by running an experiment in Architect.

If animation is enabled, each node's guiOption variable is set to TRUE at initialization. All calls to GUI API functions should be wrapped as follows:

```
if (node->guiOption) {
   // GUI API function
}
```

Figure 5-10 shows an example of calling a GUI API function. Function PhyAbstractTransmissionEnd is implemented in EXATA_HOME/libraries/wireless/src/phy_abstract.cpp.

```
void PhyAbstractTransmissionEnd(Node *node, int phyIndex) {
    PhyData* thisPhy = node->phyData[phyIndex];
    PhyDataAbstract* phy_abstract = (PhyDataAbstract *)thisPhy->phyVar;
    int channelIndex;

    PHY_GetTransmissionChannel(node, phyIndex, &channelIndex);

    assert(phy_abstract->mode == PHY_TRANSMITTING);

    phy_abstract->txEndTimer = NULL;
    //GuiStart
    if (node->guiOption == TRUE) {
        GUI_EndBroadcast(node->nodeId,
                         GUI_PHY_LAYER,
                         GUI_DEFAULT_DATA_TYPE,
                         thisPhy->macInterfaceIndex,
                         getSimTime(node));
    }
    //GuiEnd

    PHY_StartListeningToChannel(node, phyIndex, channelIndex);
    ...
 }
```

FIGURE 5-10. Calling GUI Functions

Most of the animation functions available represent "semantic" events, such as "packet was sent", which can be displayed in different ways depending on the situation. For example, "sending packet wirelessly" will look different from "sending packet over a wire", even though both call the same function.

The following general types of animation are available in EXata:

- Packet animation: sends, receives
- Queue animation: insertions, deletions, drops
- Wireless animation: transmissions, directional transmissions, antenna patterns
- Node animation: icons, labels, motion, orientation
- Statistics: definition, data updates
- Logical link: applications, abstract linkages

Table 5-10 lists the commands available in each class of animation. Many of these functions require `GuiLayers` as a parameter. This parameter is used to enable animation filtering in Architect. A programmer can take advantage of this to display only the user's own defined animation. There is no

animation defined at the GUI_CHANNEL_LAYER, so the programmer can specify this layer for all the custom animation, and disable all other layers in Architect.

TABLE 5-10. GUI API Functions for Animation

Animation Type	GUI API Functions
Packet Animation	GUI_Drop GUI_Broadcast GUI_EndBroadcast GUI_Multicast GUI_Unicast GUI_Receive
Queue Animation	GUI_AddInterfaceQueue GUI_QueueInsertPacket GUI_QueueDropPacket GUI_QueueDequeuePacket
Wireless Animation	GUI_Collision GUI_SetPatternIndex GUI_SetPatternAndAngle GUI_Broadcast GUI_EndBroadcast
Node Animation	GUI_MoveNode GUI_SetNodeOrientation GUI_SetNodeIcon GUI_SetNodeLabel GUI_SetNodeRange
Statistics Animation	GUI_DefineMetric GUI_SendIntegerData GUI_SendUnsignedData GUI_SendRealData
Logical Link Animation	GUI_AddLink GUI_DeleteLink GUI_AddApplication GUI_DeleteApplication

Programmers can customize the appearance of some of the animation using the GUI_SetEffect function. Table 5-11 lists the customizable GUI events and effects that can be customized.

TABLE 5-11. Customizable GUI Effects

GUI Event	GUI Effects
GUI_Receive	GUI_FLASH_X GUI_CIRCLE_NODE GUI_DEFAULT_EFFECT
GUI_Unicast	GUI_FLASH_X GUI_CIRCLE_NODE GUI_DEFAULT_EFFECT
GUI_Drop	GUI_FLASH_X GUI_CIRCLE_NODE GUI_DEFAULT_EFFECT

5.2.3 Adding Dynamic Statistics

EXata Architect has the capability of displaying some statistics as they change during the model execution. User-defined dynamic statistics can be added to any protocol at any layer. In this section, we describe how to add dynamic statistics to an Application Layer traffic-generating protocol. Section 4.2.5 describes how to add a traffic-generating Application Layer protocol to EXata. This section gives details of additional steps that are required to add dynamic statistics to an Application Layer protocol.

The following list summarizes the actions that need to be performed for adding dynamic statistics to an Application Layer protocol, MYPROTOCOL. Each of these steps is described in detail in subsequent sections.

1. Define handles for statistic variables in the protocol data structure (see Section 5.2.3.1).
2. Initialize the statistic handles in the protocol initialization function (see Section 5.2.3.2).
3. Modify the Application Layer dynamic statistics function APP_RunTimeStat to call MYPROTOCOL's function to send the intermediate values of the statistics to the GUI (see Section 5.2.3.3).
4. Write MYPROTOCOL's dynamic statistics function to send the intermediate values of the statistics to the GUI (see Section 5.2.3.4).

5.2.3.1 Defining Statistic Handles

An integer handle is associated with each statistic variable, which is used to send intermediate values of the statistic to the GUI. The handle for the statistic variable is included in the same data structure as the statistic variable itself. The following convention is used to declare a statistic handle:

```
int <statisticName>Id;
```

where `<statisticName>` is a statistic variable declared previously in the data structure. For example, Figure 5-11 shows a sample statistics declaration for MYPROTOCOL.

```
typedef struct {
    int BytesSent;            /* stat variable for num of bytes sent */
    int BytesSentId;          /* statistic handle */
    int BytesReceived;        /* stat variable for num of bytes received */
    int BytesReceivedId;      /* statistic handle */
} MyprotocolStatsType;
```

FIGURE 5-11. Declaring Statistic Handles

5.2.3.2 Initializing Statistic Handles

Statistic handles are initialized in the protocol's initialization function. The API GUI_DefineMetric is used to assign a unique value to each statistic handle. The parameters for GUI_DefineMetric are described below. The function GUI_DefineMetric and the enumeration types `GuiLayers`, `GuiDataTypes`, and `GuiMetrics` are declared in gui.h.

```
int GUI_DefineMetric(char*        name,
                     NodeId       nodeID,
                     GuiLayers    layer,
                     int          linkID,
                     GuiDataTypes datatype,
                     GuiMetrics   metrictype)
```

Parameters:

- `name`: Description label of the statistic in the GUI.
- `nodeID`: Node's identifier.
- `layer`: Layer at which the protocol resides. It can be one of the values enumerated in `GuiLayers`.
- `linkID`: Application session identifier. This is set to zero if it is not applicable.
- `datatype`: Statistic's data type. It can be one of the values enumerated in `GuiDataTypes`.
- `metrictype`: Indication whether the statistic is cumulative or averaged. It can be one of the values enumerated in `GuiMetrics`.

In the statistics initialization function for MYPROTOCOL, MyprotocolInitStats, initialize the statistic handles by calling GUI_MetricDefine. Figure 5-12 shows how this is done for the statistic variables and handles defined in Figure 5-11. Call MyprotocolInitStats from the MYPROTOCOL initialization function, MyprotocolInit.

```
static void MyprotocolInitStats (Node* node, MyprotocolStatsType *stats)
{
    BytesSent = 0;
    BytesReceived = 0;
    if (node->guiOption)
    {
        stats->BytesSentId = GUI_DefineMetric("Total Bytes Sent",
                                              node->nodeId,
                                              GUI_APP_LAYER,
                                              0,
                                              GUI_INTEGER_TYPE,
                                              GUI_CUMULATIVE_METRIC);
       stats->BytesReceivedId = GUI_DefineMetric("Total Bytes Received",
                                                 node->nodeId,
                                                 GUI_APP_LAYER,
                                                 0,
                                                 GUI_INTEGER_TYPE,
                                                 GUI_CUMULATIVE_METRIC);
    }
}
```

FIGURE 5-12. Initializing Statistic Handles

5.2.3.3 Modifying the Application Layer Dynamic Statistics Function

When the GUI requires intermediate statistic values, function PARTITION_PrintRunTimeStats, defined in EXATA_HOME/main/partition.cpp, is executed. This function calls the dynamic statistics functions of all layers, each of which in turn calls all protocol-specific dynamic statistics functions at that layer. For example, PARTITION_PrintRunTimeStats calls function APP_RunTimeStats to print dynamic statistics for Application Layer protocols. APP_RunTimeStat calls the function to print dynamic statistics for each protocol running at the node. APP_RunTimeStat is implemented in EXATA_HOME/main/application.cpp.

To enable dynamic statistics for the Application Layer protocol, MYPROTOCOL, modify the function APP_RunTimeStat to call MYPROTOCOL's dynamic statistics function, as shown in Figure 5-13. APP_MYPROTOCOL is the entry for MYPROTOCOL in the enumeration `AppType` (see Section 4.2.5.3) and MyprotocolRunTimeStat is the function to print MYPROTOCOL's dynamic statistics (see Section 5.2.3.4).

```
void
APP_RunTimeStat(Node *node)
{
    NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
    int i;
    AppInfo *appList = NULL;
    ...
    for (appList = node->appData.appPtr;
         appList != NULL;
         appList = appList->appNext)
    {
        /*
         * Get application specific data structure and call
         * the corresponding protocol to print the stats.
         */

        switch (appList->appType)
        {
            ...
//StartVBR
            case APP_VBR_CLIENT:
            case APP_VBR_SERVER:
            {
                VbrRunTimeStat(node, (VbrData *) appList->appDetail);
            }
                break;
//EndVBR
            ...
            case APP_MYPROTOCOL:
            {
                MyprotocolRunTimeStat(node,
                                      (MyprotocolData *) appList->appDetail);
            }
                break;
            default:
                break;
        }
    }
}
```

FIGURE 5-13. Modifying Application Layer Dynamic Statistics Function

5.2.3.4 Writing the Dynamic Statistics Function for MYPROTOCOL

To enable dynamic statistics in a model, write a function to send the intermediate values of the model's statistics to the GUI. For each statistic type (integer, double, unsigned), there is a separate API to send the value to the GUI. These APIs are listed below and their prototypes are defined in gui.h:

```
void GUI_SendIntegerData(NodeId      nodeId,
                         int         metricID,
                         int         value,
                         clocktype   time);

void GUI_SendRealData(NodeId      nodeId,
                      int         metricID,
                      double      value,
                      clocktype   time);

void GUI_SendUnsignedData(NodeId      nodeId,
                          int         metricID,
                          unsigned    value,
                          clocktype   time);
```

Parameters:

- `nodeId`: Node's identifier.
- `metricID`: Handle assigned to the statistic (see Section 5.2.3.2).
- `value`: Current value of the metric.
- `time`: Current simulation time.

Write the dynamic statistics function for MYPROTOCOL, MyprotocolRunTimeStat. Include the prototype for MyprotocolRunTimeStat in the protocol's header file, myprotocol.h. Figure 5-14 shows a sample implementation of MyprotocolRunTimeStat. The `node->guiOption` clause ensures that the protocol sends data to the GUI only if the GUI option is selected.

```
void MyprotocolRunTimeStat(Node* node, MyprotocolData* dataPtr) {
    clocktype now = getSimTime(node);
    if (node->guiOption)
    {
        GUI_SendIntegerData(node->nodeId,
                            dataPtr->stats.BytesSentId,
                            dataPtr->stats.BytesSent,
                            now);
        GUI_SendIntegerData(node->nodeId,
                            dataPtr->stats.BytesReceivedId,
                            dataPtr->stats.BytesReceived,
                            now);
    }
}
```

FIGURE 5-14. Dynamic Statistics Function for MYPROTOCOL

5.3 Customizing EXata Packet Tracer

Packet Tracer customization consists of adding tracing support for new protocols. First, Simulator code is updated to produce trace output for the new protocol (see Section 4.9). Second, a description of that trace output is made available to Packet Tracer. This section describes how to add the description of a new protocol header to files used by Packet Tracer.

5.3.1 Trace File Generated by Simulator

The syntax of a trace file produced by Simulator is described in Section 4.9.1. A trace file contains trace records and each trace record contains values of the fields of each header of the packet. As an example, Figure 5-15 shows the definition of the data structure for the IP header in file EXATA_HOME/libraries/developer/src/network_ip.h, and Figure 5-17 shows a record from a trace file generated by Simulator corresponding to the trace of an IP header.

```
typedef
struct ip_header_str
{
    UInt32 ip_v_hl_tos_len;        /* version, header length, type of
                                      service and total length */
    UInt16 ip_id;                  /* IP protcol ID */
    UInt16 ipFragment;
    unsigned char  ip_ttl;         /* time to live */
    unsigned char  ip_p;           /* protocol */
    unsigned short ip_sum;         /* checksum */
    unsigned       ip_src,ip_dst;  /* source and dest address */

} IpHeaderType;
```

FIGURE 5-15. IP Header Data Structure

```
...
<rec>
<rechdr> 1 1 10.001383416 34 1 3 <action> 4 <queue> 0 192</queue></action></rechdr>
<recbody>
<ipv4>4 5 48 0 0 44 1 <flags>0 0 0</flags> 0 1 123 0 0.0.0.1 255.255.255.255</ipv4>
</recbody>
</rec>
...
```

FIGURE 5-16. Trace Record Showing an IP Header Trace

5.3.2 Definition Files Used by Packet Tracer

Packet Tracer makes use of two files to interpret and display the data from a trace file. These files are HeaderDef.xml and Datatype.xml in the folder EXATA_HOME/gui/settings. File HeaderDef.xml contains definitions of the protocol headers and file Datatype.xml contains definitions of data types used in the protocol header definitions. As an example, Figure 5-17 shows the descriptions of the UDP and IPv4 headers in the file HeaderDef.xml and Figure 5-18 shows some of the basic data type definitions from the file Datatype.xml. Adding tracing support for a new protocol header involves editing the file HederDef.xml; in some cases, the file Datatype.xml may need to be changed as well.

```
<protocol_header_def>
...
<!-- udp -->

<protocolheader name="udp" label="UDP" length="20" type="" color="#00ffff">
    <u16 label="Source Port" />
    <u16 label="Destination Port" />
    <u16 label="Length" postlabel="bytes" />
    <u16 label="Checksum" postlabel="not computed" />
</protocolheader>
...
<!-- ipv4 -->
 ...
<protocolheader name="ipv4" label="IPv4" length="20" type="" color="#ff0000">
    <group name="flags" label="Flags" length="3">
        <u1 label="Reserved" />
        <u1 label="Don't fragment" />
        <u1 label="More fragments" />
    </group>

    <u4 label="Version" />
    <u4 label="Header Length" postlabel="32 bit words" />
    <u6 label="TOS" />
    <u1 label="ECN ECT" />
    <u1 label="ECN CE" />
    <u16 label="Total Length" />
    <u16 label="Identification" />
    <flags />
    <u13 label="Fragment Offset" />
    <u8 label="TTL" />
    <u8 label="Protocol" />
    <u16 label="Checksum" postlabel="not computed" />
    <ipv4Addr label="Source IP" />
    <ipv4Addr label="Destination IP" />
</protocolheader>
...
```

FIGURE 5-17. Header Descriptions in File HeaderDef.xml

```
<data_type>
    <!-- Unsigned types. Bitlength 1..64 -->
    <basic name="u1"    signed="false"  bitlength="1"   />
    <basic name="u2"    signed="false"  bitlength="2"   />
    <basic name="u3"    signed="false"  bitlength="3"   />
    <basic name="u4"    signed="false"  bitlength="4"   />
    ...
    <basic name="u16"   signed="false"  bitlength="16"  />
    <basic name="u17"   signed="false"  bitlength="17"  />
    ...
    <basic name="u64"   signed="false"  bitlength="64"  />

    <!-- Signed types. Bitlength 2..64 -->
    <basic name="s2"    signed="true"   bitlength="2"   />
    <basic name="s3"    signed="true"   bitlength="3"   />
    ...
    <basic name="s63"   signed="true"   bitlength="63"  />
    <basic name="s64"   signed="true"   bitlength="64"  />

    <string name="str" />

    <string name="ipv4Addr" validation= "\d{1-3}\.\d{1-3}\.\d{1-3}\.\d{1-3}" />
    <string name="macAddr" validation= "\xx-\xx-\xx-\xx-\xx-\xx" />

    <float name="double" />

    <float name="simTime" format="4.6" />

</data_type>
```

FIGURE 5-18. Data Type Definitions in File Datatype.xml

5.3.3 Packet Tracer Display

Packet Tracer uses the definitions of XML elements in the file HeaderDef.xml (see Figure 5-17) and the definition of data types in the file Datatype.xml (see Figure 5-18) to parse and display the data in trace files. Each record in the trace file has a `rechdr` and a `recbody` (see Section 4.9.1). Packet Tracer parses the data contained in a `rechdr` of a trace according to the `record_header` element in file HeaderDef.xml. Packet Tracer displays each `rechdr` as a row in the trace table, as shown in Figure 5-19. When a row is selected, the details of the action that triggered the generation of the `rechdr` are displayed below the trace table. If the row corresponds to a queue action (e.g., enqueue or dequeue), then the queue identifier and priority are also displayed below the trace table.

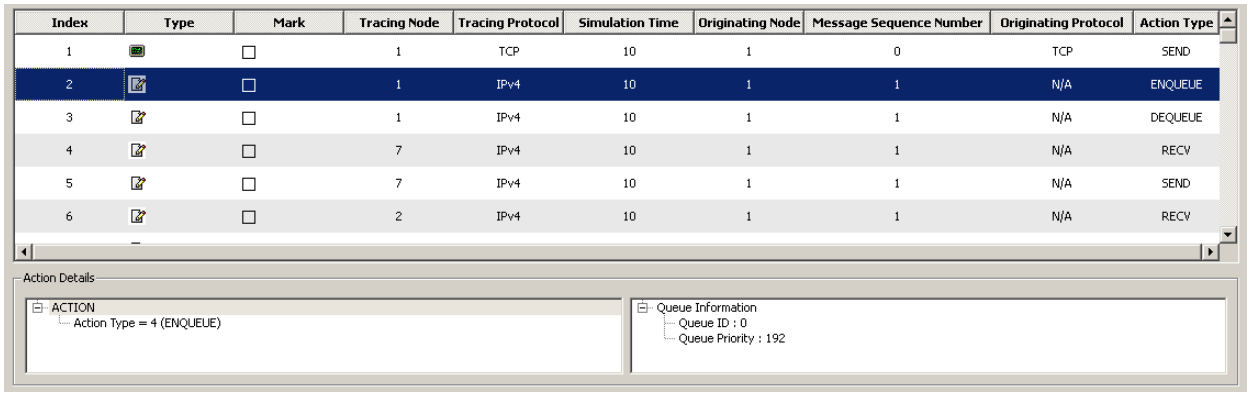


FIGURE 5-19. Trace Table Displayed by Packet Tracer

The `recbody` of a trace record contains one or more protocol headers. Packet Tracer uses the definition of `protocolheader` elements in the file HeaderDef.xml to parse and display the data contained in `recbody`. Based on the IP header description in HeaderDef.xml (.i.e., the `protocolheader` element whose `name` attribute is `"ipv4"`) and the definition of the data types in file in Datatype.xml, Packet Tracer parses the IP header data in Figure 5-16 as follows:

- `Version` is assigned the value 4
- `Header Length` is assigned the value 5
- `TOS` is assigned the value 48
- `ECN ECT` is assigned the value 0
- `ECN CE` is assigned the value 0
- `Total Length` is assigned the value 44
- `Identification` is assigned the value 1
- `Reserved` is assigned the value 0
- `Don't Fragment` is assigned the value 0
- `More Fragments` is assigned the value 0
- `Fragment Offset` is assigned the value 0
- `TTL` is assigned the value 1
- `Protocol` is assigned the value 123
- `Checksum` is assigned the value 0
- `Source IP` is assigned the value 0.0.0.1
- `Destination IP` is assigned the value 255.255.255.255

Packet Tracer displays the header data in the Explorer window. Figure 5-20 shows the display corresponding to the IP header data in Figure 5-16.

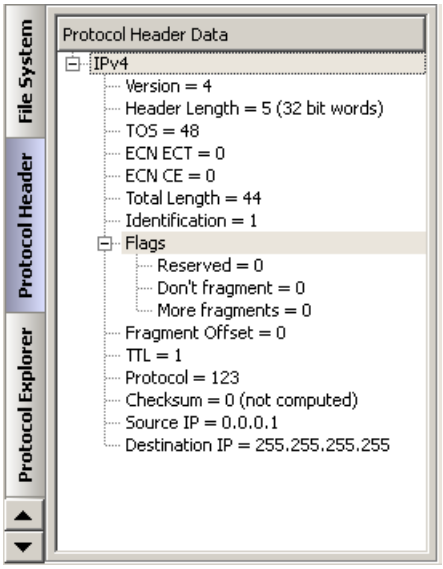


FIGURE 5-20. Display of an IP Header in Packet Tracer

5.3.4 Adding Trace Capability for a New Header

To enable Packet Tracer to recognize and display a new header, add the description of the new header to the file EXATA_HOME/gui/settings/HeaderDef.xml. The header description in HeaderDef.xml should match the format of the header trace in the trace file (see Section 4.9.1).

The format of a protocol header in a trace file is specified by means of a `protocolheader` element in the file HeaderDef.xml. A `protocolheader` element uses *data type definitions* and *data display definitions*. This section first describes data type definitions and data display definitions and then describes the syntax of a `protocolheader` element.

5.3.4.1 Data Type Definitions

Data type definitions are used to declare data types used in the description of protocol headers. Data type definitions are specified in the file EXATA_HOME/gui/settings/Datatype.xml. A data type definition can also be included in the protocol header description itself as a child element of a `protocolheader` element (see Section 5.3.4.3).

This section describes the format of data type definition elements that can be used in protocol header descriptions.

5.3.4.1.1 The `basic` Data Type

The `basic` element is used for defining integer types. Table 5-12 describes the attributes of the `basic` element. The `basic` element does not have any children.

TABLE 5-12. Attributes of the `basic` Element

Attribute	Required or Optional	Description of the Attribute
`name`	Required	Name of the type definition.
`signed`	Required	Indication whether the number can be negative. • `signed="true"` indicates the number can be negative. • `signed="false"` indicates the number can only be non-negative.
`bitlength`	Required	Number of bits in the binary representation of the number.

The following are examples of the `basic` element:

```
<basic name="u12"   signed="false"  bitlength="12"  />
<basic name="s23"   signed="true"   bitlength="23"  />
```

5.3.4.1.2 The `float` Data Type

The `float` element is used for defining floating point types. Table 5-13 describes the attributes of the `float` element. The `float` element does not have any children.

TABLE 5-13. Attributes of the `float` Element

Attribute	Required or Optional	Description of the Attribute
`name`	Required	Name of the type definition.
`format`	Optional	Not used currently.

The following are examples of the `basic` element:

```
<float name="double" />
<float name="simTime" format="4.6" />
```

5.3.4.1.3 The `char` and `string` Data Types

The `char` and `string` elements are used for defining character data types. Table 5-14 describes the attributes of the `char` and `string` elements. The `char` and `string` elements do not have any children.

TABLE 5-14. Attributes of the `char` and `string` Elements

Attribute	Required or Optional	Description of the Attribute
`name`	Required	Name of the type definition.
`validation`	Optional	Not used currently.
`length`	Optional	Not used currently.

The following are examples of the `char` and `string` elements:

```
<char name="c" length="1" />
<string name="str" /
<string name="macAddr" validation= "\xx-\xx-\xx-\xx-\xx-\xx" />
```

5.3.4.1.4 The `enum` Data Type

The `enum` element is used for defining an enumerated mapping between strings and integers. Table 5-15 describes the attributes of the `enum` element. The `enum` element has `enumitem` element as its child.

TABLE 5-15. Attributes of the `enum` Element

Attribute	Required or Optional	Description of the Attribute
`name`	Required	Name of the type definition.
`label`	Required	Name displayed by Packet Tracer.
`format`	Optional	Not used currently.
`length`	Optional	Used to validate the element type. If it is not specified, the default value is 32.

The following is an example of the `enum` element:

```
<enum name="actiontype" label="Action type" length="8">
     <enumitem postlabel="SEND" enumvalue="1" />
     <enumitem postlabel="RECV" enumvalue="2" />
     <enumitem postlabel="DROP" enumvalue="3" />
     <enumitem postlabel="ENQUEUE" enumvalue="4" />
     <enumitem postlabel="DEQUEUE" enumvalue="5" />
</enum>
```

The `enumitem` element is used in the definition of an `enum` type. Table 5-16 describes the attributes of the `enumitem` element. The `enumitem` element has no children.

TABLE 5-16. Attributes of the `enumitem` Element

Attribute	Required or Optional	Description of the Attribute
`postlabel`	Required	String displayed by Packet Tracer after the item's value.
`enumvalue`	Required	Integer value displayed by Packet Tracer for the item.

The following are examples of the `enumitem` element:

```
<enumitem postlabel="Hello" enumvalue="0" />
<enumitem postlabel="Database Desc." enumvalue="1" />
```

5.3.4.1.5 The group Data Type

The group element is used for defining a structured listing of data definitions. Table 5-17 describes the attributes of the group element. The group element can have data type definition elements (basic, float, char, string, enum, and group) and data display definition elements (see Section 5.3.4.2) as its children. The data type definition elements must be specified before the data display definition elements.

TABLE 5-17. Attributes of the group Element

Attribute	Required or Optional	Description of the Attribute
name	Required	Name of the type definition.
label	Required	Name displayed by Packet Tracer.
length	Optional	Not used currently.
contents_repeat	Optional	Indication whether children of the group element are repeated. • contents_repeat="yes" indicates the children of the group are repeated an indeterminate number of times. • contents_repeat="no" indicates the children of the group appear only once.

The following is an example of the group element:

```
<group name="flags" label="Flags" length="3">
   <u1 label="Reserved" />
   <u1 label="Don't fragment" />
   <u1 label="More fragments" />
</group>
```

5.3.4.2 Data Display Definitions

Data display definitions determine how data read from a trace file is displayed by Packet Tracer. Data display definition elements are specified as children of a group element (see Section 5.3.4.1.5) or a protocolheader element (see Section 5.3.4.3). The element name of a data display definition element is the name attribute of a data type definition element. This data type definition element should either be a child of the parent group or protocolheader element or should be defined in the file Datatype.xml. The attributes of a data display definition element are listed in Table 5-18.

TABLE 5-18. Attributes of the Data Display Definition Element

Attribute	Required or Optional	Description of the Attribute
label	Optional	String displayed by Packet Tracer before the element's value.
postlabel	Optional	String displayed by Packet Tracer after the element's value.

The following are examples of the data display definition element:

```
<u4 label="Version" />
<u4 label="Header Length" postlabel="32 bit words" />
```

5.3.4.3 Protocol Header Definitions

A protocol header format is described by means of a `protocolheader` element in the file HeaderDef.xml. A `protocolheader` element determines how Packet Tracer interprets the `recbody` element of a trace file. The attributes of the `protocolheader` element are listed in Table 5-19.

TABLE 5-19. Attributes of the `protocolheader` Element

Attribute	Required or Optional	Description of the Attribute
`name`	Required	Protocol name as it appears in the trace file.
`label`	Required	Protocol name displayed by Packet Tracer.
`length`	Optional	Not used currently.
`type`	Required	Indication of the packet type. It can have one of the following four values: "`application`", "`control`", "`fragment`", or "". This determines the icon displayed in the second column (labelled "Type") in the trace table. If an empty string ("") is specified, the same icon as is used for "`application`" is displayed.
`color`	Optional	Not used currently.

The children of the `protocolheader` element are data type definition elements (see Section 5.3.4.1) and data display definition elements (see Section 5.3.4.2). Within a `protocolheader` element, all data type definitions are listed before data display definitions.

Figure 5-17 shows examples of the `protocolheader` element.

6 Interfacing with EXata: External Interface API

The External Interface API allows EXata to interact with external entities such as other programs or physical devices. Figure 6-1 illustrates the responsibilities of the interface developer and how the External Interface API operates with EXata.

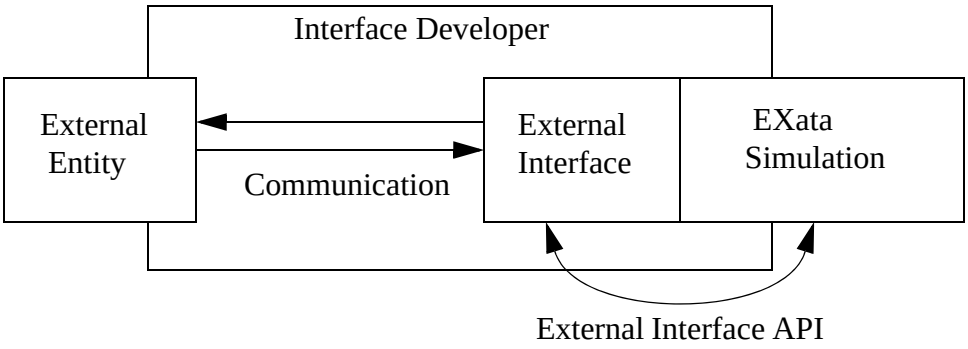


FIGURE 6-1. External Interface API

The *External Entity* exists outside of the EXata simulation. It is the responsibility of the *External Interface* to implement the communication tasks with the External Entity. The External Interface interacts with the EXata Simulation using the *External Interface API*.

The API is divided into two main sections: interface registration functions and utility functions. Interface registration functions allow the interface developer to create an interface and define how it operates. The utility functions simplify interface development by automating routine tasks.

In this chapter we describe how to develop an external interface for EXata. We give an example of an external interface in Section 6.1. Section 6.2 describes the interface registration functions, and the utility functions are described in Section 6.3.

6.1 Tutorial

We illustrate the process of developing an external interface for EXata by means of a tutorial. This tutorial describes the steps in writing an external interface that communicates with an external program through a socket and interacts with an Application Layer protocol in EXata.

This tutorial has three components:

- TUTORIALTESTER: This program corresponds to the external entity of Figure 6-1, and is described in Section 6.1.1.
- INTERFACETUTORIAL: This is an Application Layer program in EXata that interacts with TUTORIALTESTER through an external interface, and is described in Section 6.1.2.
- External Interface: This is the external interface used by INTERFACETUTORIAL to interact with TUTORIALTESTER, and is described in Section 6.1.3.

6.1.1 The TUTORIALTESTER Program

The first step in writing an external interface is to consider the external entity that the interface is meant to interact with. The external entity may be a program running on the same computer, a program running on a different computer, a hardware device, or something entirely different. Each type of external entity interacts with EXata in a different manner. The interface developer should tailor the interface to the external entity.

In this tutorial, the external entity is a simple interactive program called TUTORIALTESTER. The source code for the TUTORIALTESTER program can be found in the file EXATA_HOME/scenarios/interfacetutorial/tutorialtester.cpp.

The TUTORIALTESTER program may be built on UNIX by executing the following command:

```
g++ -o tutorialtester tutorialtester.cpp
```

The TUTORIALTESTER program may be built on Windows using the following command:

```
cl tutorialtester.cpp /link ws2_32.lib
```

The TUTORIALTESTER program is an interactive console application that accepts user input in the following format:

```
<source node id> <destination node id> <command>
```

where

`<source node id>` : Source node identifier

`<destination node id>` : Destination node identifier

`<command>` : Command, which can be either a *set* command or a *get* command

A set command has the following format:

```
s <value>
```

where

`<value>` : Any string

A get command has the following format:

```
g
```

TUTORIALTESTER interacts with the Application Layer program INTERFACETUTORIAL in the EXata simulation. Each node in the EXata simulation that has an INTERFACETUTORIAL application running on it maintains a variable. The set command acts on this variable by assigning it a new value, and the get command retrieves the variable's current value and sends it back to the TUTORIALTESTER program. TUTORIALTESTER accepts input from a user and sends the input to EXata through a socket. This results in the INTERFACETUTORIAL program running at the source node sending the command contained in the user input to the destination node.

6.1.2 The INTERFACETUTORIAL Application Layer Protocol

The INTERFACETUTORIAL application is implemented by the files interfacetutorial_app.h and interfacetutorial_app.cpp in the directory EXATA_HOME/interfaces/interfacetutorial/src. The commented source code can be read from those files. This section provides a summary of the INTERFACETUTORIAL functions.

AppInterfaceTutorialGet: This function searches the list of applications running at a node for the INTERFACETUTORIAL application. This is typically called from the message processing function to retrieve the application data structure.

AppInterfaceTutorialNew: This function allocates a new INTERFACETUTORIAL application data structure. This function is called from AppInterfaceTutorialInit.

AppLayerInterfaceTutorial: This function processes Application Layer messages, such as incoming UDP packets. This is the heart of the application. Each received command is analyzed here. For set commands, the application's variable is updated to the value specified in the command. For get commands, the value of the application's variable is forwarded back to the external interface where it is sent to the external entity.

AppInterfaceTutorialInit: This function initializes a new INTERFACETUTORIAL application.

AppInterfaceTutorialFinalize: This function finalizes the application. There are no tasks to perform as this is a simple application. Typically a protocol would print statistics in the finalization function.

The INTERFACETUTORIAL application must be integrated into EXata. The procedure is similar to adding an Application Layer protocol to EXata (see Section 4.2.5) and is summarized below:

- Add the APP_INTERFACETUTORIAL enumeration value to the `AppType` enumeration in EXATA_HOME/include/application.h.
- Add the TRACE_INTERFACETUTORIAL enumeration value to the `TraceProtocolType` enumeration in EXATA_HOME/include/trace.h.

- Call the AppInterfaceTutorialInit function from the APP_InitializeApplications function, which is defined in EXATA_HOME/main/application.cpp.
- Call the AppLayerInterfaceTutorial function from the APP_ProcessEvent function, which is defined in EXATA_HOME/main/application.cpp.
- Call the AppInterfaceTutorialFinalize function from the APP_Finalize function, which is defined in EXATA_HOME/main/application.cpp.
- Compile the application into EXata. In this case the application is packaged as an addon. Edit the addons makefile for your platform. If you are running EXata on Windows, uncomment the following line in the file EXATA_HOME/main/Makefile-addons-windows:

  ```
  include ../interfaces/interfacetutorial/Makefile-windows
  ```

  If you are running EXata on UNIX, uncomment the following line in the file EXATA_HOME/main/Makefile-addons-unix:

  ```
  include ../interfaces/interfacetutorial/Makefile-unix
  ```

  See Section 4.10 for instructions to activate an addon module.
- In the configuration file, specify INTERFACETUTORIAL as the application running at each node that is capable of sending and receiving commands from the external interface.

  Sample configuration files are included in the directory EXATA_HOME/scenarios/interfacetutorial. To run EXata using the sample configuration files, copy the files tutorial.config, tutorial.nodes and tutorial.app from EXATA_HOME/scenarios/interfacetutorial into the directory EXATA_HOME/bin, and run EXata using tutorial.config as the configuration file.

6.1.3 The Interface Tutorial External Interface

The external interface for this tutorial is implemented by the files interfacetutorial.h and interfacetutorial.cpp in the directory EXATA_HOME/interfaces/interfacetutorial/src. This section provides a summary of the tutorial's external interface functions.

InterfaceTutorialInitializeNodes: This function initializes the external interface. It is called after all nodes and protocols have been created. It creates an interface-specific data structure and opens TCP port 5132 for listening. The TUTORIALTESTER program connects to this TCP port and sends in commands.

InterfaceTutorialReceive: This function receives user input from a socket. It parses the input, determines the source node, destination node, and the command. It sends the command from source node to destination node by calling EXTERNAL_SendDataAppLayerUDP.

InterfaceTutorialForward: This function sends information back to the TUTORIALTESTER program through the TCP socket. This function is called when a destination node receives the "get" command.

InterfaceTutorialFinalize: This function is called at the end of the simulation and closes the open socket connections.

Like the INTERFACETUTORIAL application, the external interface must be integrated into EXata. The following list summarizes the steps required to integrate the external interface:

- Register the interface by calling the function EXTERNAL_RegisterExternalInterface from the EXTERNAL_UserFunctionRegistration function, which is defined in EXATA_HOME/main/external.cpp (see Section 6.2).

- Register the interface's functions by calling the function EXTERNAL_RegisterFunction from the EXTERNAL_UserFunctionRegistration function, which is defined in EXATA_HOME/main/external.cpp (see Section 6.2).
- Compile the source file, EXATA_HOME/interfaces/interfacetutorial/src/interfacetutorial.cpp, into EXata. To do this, edit the addons makefile for your platform, as described in Section 6.1.2.

6.2 Interface Registration

This section describes the functions to register an external interface and the callback functions implemented by the interface (see Section 6.2.1) and the format of the callback functions (see Section 6.2.2).

6.2.1 Registration Functions

Function EXTERNAL_UserFunctionRegistration, which is defined in EXATA_HOME/main/external.cpp, is called by the kernel at the beginning of the simulation. This function is used to register an external interface (by calling the function External_RegisterExternalInterface) and the callback functions implemented by the interface (by calling the function External_RegisterFunction).

- **External_RegisterExternalInterface**: This function registers a new external interface with EXata and creates the necessary data structures. This function must be called before any other function that requires an EXTERNAL_Interface* argument.

  Syntax:

  ```
  EXTERNAL_Interface* EXTERNAL_RegisterExternalInterface(
       EXTERNAL_InterfaceList *list,
       char *name,
       EXTERNAL_PerformanceParameters params,
       ExternalInterfaceType type);
  ```

  Parameters:

  `list` : The list of external interfaces

  `name` : The name of the external interface

  `params` : The performance parameters. Currently no performance parameters are supported, so pass EXTERNAL_NONE.

  `type` : The type of the external interface

  Return value:

  A pointer to the newly registered external interface

- **EXTERNAL_RegisterFunction**: This function registers a callback function for an external interface. Callback functions are described in Section 6.2.2.

  Syntax:

  ```
  void EXTERNAL_RegisterFunction(
       EXTERNAL_Interface *iface,
       EXTERNAL_FunctionType type,
       EXTERNAL_Function function);
  ```

Parameters:

`iface` : The interface structure

`type` : The function type (`EXTERNAL_INITIALIZE`, `EXTERNAL_INITIALIZE_NODES`, `EXTERNAL_TIME`, `EXTERNAL_SIMULATION_HORIZON`, `EXTERNAL_PACKET_DROPPED`, `EXTERNAL_RECEIVE or EXTERNAL_FORWARD`, `EXTERNAL_FINALIZE`)

`function` : The pointer to the function that is called by EXata

Once the interface and its callback functions have been registered EXata starts using the interface. EXata calls the registered callback functions as needed.

6.2.2 Callback Functions

The interface developer determines the behavior of the external interface by providing callback functions that perform certain tasks. There are eight callback functions the interface developer may provide. An external interface may not need all eight callback functions, in which case the interface needs to implement and register only the functions that it needs. EXata calls the registered functions as necessary.

The callback functions are:

- **Initialize**: This is an initialization function called before nodes and protocols are created. This function is used for setting up data structures and initializing services that are used on a simulation-wide basis. For example, an HLA Initialize function would initialize the RTI and Federate ambassadors, and a network emulation Initialize function would initialize the packet sniffing library.

  Syntax:

  ```
  void InterfaceInitializeFunction(
       EXTERNAL_Interface *iface,
       NodeInput *nodeInput)
  {
       // Create socket connections, open log files, etc.
  }
  ```

  Parameters:

  `iface` : The interface structure

  `nodeInput` : The NodeInput structure. Contains experiment configuration information.

- **InitializeNodes**: This is an initialization function called after nodes and protocols have been created. This function is called immediately before the simulation begins. This function is used for setting up individual nodes or protocols to operate with the external interface. Additionally, the interface might initialize data used for time management (such as setting up a correspondence between External Time and Simulation time). For example, an HLA InitializeNodes function would create an HLA application on each node that is associated with an HLA object, and a network emulation InitializeNodes function would create correspondences between real network devices and simulated nodes, and possibly create a protocol at the Application Layer to handle events.

  Syntax:

  ```
  void InterfaceInitializeNodesFunction(
      EXTERNAL_Interface *iface,
      NodeInput *nodeInput)
  {
       // Conduct further initialization.
  }
  ```

Parameters:

`iface` : The interface structure

`nodeInput` : The NodeInput structure. Contains experiment configuration information.

- **Time**: This function is called to query the time of the external interface.

  Syntax:

```
clocktype InterfaceTimeFunction(EXTERNAL_Interface *iface)
{
    // Compute and return the time of the external entity.
}
```

  Parameters:

  `iface` : The interface structure

  Return value:

  The current time of the external entity in nanoseconds. 0 corresponds to the start of the simulation.

- **SimulationHorizon**: This function is called by the kernel to query the external interface's simulation horizon. The value of the simulation horizon controls the advance of the simulation clock. The interface increases the horizon to indicate that it is ok for the simulation clock to advance. The kernel executes events less than and up to the horizon. Once the simulation reaches the horizon, it executes a loop in which it calls this function and the Receive function (explained below) until the horizon is again advanced to allow the execution of more events.

  Syntax:

```
void InterfaceSimulationHorizonFunction(EXTERNAL_Interface *iface)
{
    // Set iface->horizon to the external entity's simulation horizon.
}
```

  Parameters:

  `iface` : The interface structure

- **PacketDropped:** This function is called when a packet is dropped. Since there are many places a packet can be dropped, support for this function is not included by default. It is the interface developer's responsibility to add support for this callback.

  Syntax:

```
void InterfacePacketDroppedFunction(EXTERNAL_Interface *iface)
{
    // Process the dropped packet.
}
```

  Parameters:

  `iface` : The interface structure

- **Receive:** This function retrieves information from the external interface and injects messages into EXata.

  Syntax:

```
void InterfaceReceiveFunction(EXTERNAL_Interface *iface)
{
     // Receive data from external entity and add messages to
     // EXata simulation.
}
```

  Parameters:

  `iface` : The interface structure

- **Forward:** This function forwards information back to the external source. It is essentially the reverse of the Receive function.

  Syntax:

```
void InterfaceForwardFunction(
     EXTERNAL_Interface *iface,
     void *forwardData,
     int forwardSize)
{
     // Send data back to external entity.
}
```

  Parameters:

  `iface` : The interface structure

  `forwardData` : A pointer to the data that needs to be forwarded. Since the external interface originally created this data it is expected that the external interface knows how to interpret the data.

  `forwardSize` : The size in bytes of the forwarded data

- **Finalize:** This function is called at the end of the simulation. It is used for freeing memory and stopping the services started by the external interface.

  Syntax:

```
void InterfaceFinalizeFunction(EXTERNAL_Interface *iface)
{
     // Close socket connections, close log files, free memory, etc.
}
```

  Parameters:

  `iface` : The interface structure

6.3 Utility Functions

The External Interface API provides several utility functions that simplify the job of the interface developer. The utility functions are divided into three categories:

- External interface API utility functions
- Functions for injecting traffic from external interfaces
- Operating system-specific utility functions for sockets

6.3.1 External Interface API Utility Functions

The file EXATA_HOME/include/WallClock.h defines a C++ class, `WallClock`, which is used to keep track of elapsed real time during a simulation. The class implements methods for pausing and resuming the wall clock and for reading the wall clock's value. See file EXATA_HOME/include/WallClock.h or *API Reference Guide* for a description of the `WallClock` class and its methods.

Additional external interface API utility functions are defined in the files external.h, external_util.h, and partition.h in the folder EXATA_HOME/include.

- **EXTERNAL_SetTimeManagementRealTime**: This function automates synchronizing EXata with real time. This function turns time management on and specifies the *lookahead* value. Internally, this function registers a horizon function that uses real time for the horizon value. The lookahead value is the amount of time that EXata is allowed to run ahead of real time. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_SetTimeManagementRealTime(
         EXTERNAL_Interface *iface,
         clocktype lookahead);
  ```

  Parameters:

  `iface` : The external interface

  `lookahead` : How far into the future the simulation is allowed to run

- **EXTERNAL_ChangeRealTimeLookahead**: This function modifies the lookahead value. This function can be called only after EXTERNAL_SetTimeManagementRealTime. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_ChangeRealTimeLookahead(
         EXTERNAL_Interface *iface,
         clocktype lookahead);
  ```

  Parameters:

  `iface` : The external interface

  `lookahead` : The new lookahead value

- **EXTERNAL_QueryExternalTime**: This function returns the time of an external interface using the interface's time callback function (see Section 6.2.2). This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  clocktype EXTERNAL_QueryExternalTime(EXTERNAL_Interface *iface);
  ```

  Parameters:

  `iface` : The external interface

  Return Value:

  The time of the external interface. The function returns `EXTERNAL_MAX_TIME` if the time callback function is not defined for the interface.

- **EXTERNAL_QuerySimulationTime**: This function returns the current simulation time. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  clocktype EXTERNAL_QuerySimulationTime(EXTERNAL_Interface *iface);
  ```

  Parameters:

  `iface` : The external interface

  Return Value:

  The current simulation time

- **EXTERNAL_QueryRealTime**: This function returns the wall clock time in the EXata time format (in nanoseconds). This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  clocktype EXTERNAL_QueryRealTime();
  ```

  Return Value:

  The real (wall clock) time

- **EXTERNAL_QueryCPUTime**: This function returns the amount of CPU time used by EXata. The first call to this function by an interface returns 0; subsequent calls to this function by the same interface returns the amount of CPU time used since the first call to the function. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  clocktype EXTERNAL_QueryCPUTime(EXTERNAL_Interface *iface);
  ```

  Parameters:

  `iface` : The external interface

  Return Value:

  0, for the first call to the function; CPU time used since the first call, for subsequent calls

- **EXTERNAL_Sleep**: This function puts EXata to sleep for a specified length of time. Depending on the platform that is being used, the length of time spent sleeping could be greater. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_Sleep(clocktype amount);
  ```

  Parameters:

  `amount` : The amount of time to sleep

- **EXTERNAL_SetReceiveDelay**: This function sets the minimum delay between two consecutive calls to the receive function (see Section 6.2.2). This prevents the receive function from being called too frequently, which may adversely affect performance. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_SetReceiveDelay(
         EXTERNAL_Interface *iface,
         clocktype delay);
  ```

  Parameters:

  `iface` : The external interface

  `delay` : The minimum delay

- **EXTERNAL_ForwardData**: This function sends data back to the external source with no time stamp. The forward function (see Section 6.2.2) receives this data and processes it. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_ForwardData(
         EXTERNAL_Interface *iface,
         Node *node,
         void *forwardData,
         int forwardSize,
         EXTERNAL_ForwardData_ReceiverOpt FwdReceiverOpt =
           EXTERNAL_ForwardDataAssignNodeID_On);
  ```

  Parameters:

`iface`	: The external interface
`node`	: The node that is forwarding the data
`forwardData`	: The data to forward
`forwardSize`	: The size of the data to forward
`FwdReceiverOpt`	: Option whether to store the receiving node for forwarded data

- **EXTERNAL_RemoteForwardData**: This function is similar to EXTERNAL_ForwardData except that it sends data back to the external interface that is on a different partition. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  void EXTERNAL_RemotForwardData(
         EXTERNAL_Interface *iface,
         Node* node,
         void *forwardData,
         int forwardSize
         int partitionId
         clocktype delay);
  ```

  Parameters:

`iface`	: The external interface
`node`	: The node that is forwarding the data
`forwardData`	: The data to forward
`forwardSize`	: The size of the data to forward
`partitionId`	: The identifier of the partition that the external interface is on
`delay`	: The delay for forwarding the data

- **EXTERNAL_GetInterfaceByName**: This function searches an interface list for an interface with the specified name. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

  ```
  EXTERNAL_Interface* EXTERNAL_GetInterfaceByName(
         EXTERNAL_InterfaceList *list,
         char *name);
  ```

Parameters:

`list` : The external interface list

`name` : The interface's name

Return Value:

The interface, if found; `NULL`, if not found

- **EXTERNAL_Bootstrap**: This function is called by the kernel early in the simulation initialization process (before threads are created) and provides a place for the external interface code to examine the simulation command line arguments. This function is defined in EXATA_HOME/include/external.h.

  Syntax:

```
void EXTERNAL_Bootstrap(
       int argc,
       char * argv [],
       SimulationProperties * simProps,
       PartitionData * partitionData);
```

  Parameters:

  `argc` : The number of arguments in the command line

  `argv` : The arguments in the command line

  `simProps` : Global properties of the simulation for all partitions

  `partitionData` : Pointer to the data for this partition

  Note **If you wish to add new command line options for use by your external interface, you must update the function PARTITION_ParseArgv in the file EXATA_HOME/main/partition.cpp to include checks for the new options; otherwise, that function will output an unknown option error and the simulator will exit.**

- **EXTERNAL_SetSimulationEndTime**: This function sets the end time for the simulation. This function is defined in EXATA_HOME/include/external_util.h.

  Syntax:

```
void EXTERNAL_SetSimulationEndTime(
     EXTERNAL_Interface* iface,
     clocktype endTime = 0);
```

  Parameters:

  `iface` : The external interface

  `endTime` : The time at which the simulation should end. If this parameter is omitted, the simulation ends at the current time.

- **PARTITION_ClientStateSet**: This function sets or replaces a pointer to the specified client's state in the indicated partition. This function is defined in EXATA_HOME/include/partition.h.

  Syntax:

```
void PARTITION_ClientStateSet(
        PartitionData* partitionData,
        const char*    stateName,
        void*          clientState);
```

Parameters:

`partitionData` : Pointer to the data for this partition
`stateName` : The name used to locate the client's state information
`clientState` : Pointer to the data structure the client wishes to store

- **PARTITION_ClientStateFind**: This function searches for the specified client's state in the indicated partition and returns a pointer to it. This function is defined in EXATA_HOME/include/partition.h.

  Syntax:

  ```
  void* PARTITION_ClientStateFind(
          PartitionData* partitionData,
          const char*    stateName)
  ```

  Parameters:

  `partitionData` : Pointer to the data for this partition
  `stateName` : The name used to locate the client's state information

  Return Value:

  A pointer to the client's state. If the state is not found, the function returns `NULL`.

- **PARTITION_GetRealTimeMode**: This function checks whether the simulation is executing in real time mode. This function is defined in EXATA_HOME/include/partition.h.

  Syntax:

  ```
  bool
  PARTITION_GetRealTimeMode (PartitionData * partitionData);
  ```

  Parameters:

  `partitionData` : Pointer to the data for this partition

  Return Value:

  `TRUE`, if the simulation is running in real time mode; `FALSE`, otherwise.

- **PARTITION_SetRealTimeMode**: This function sets the simulation execution mode. This function is defined in EXATA_HOME/include/partition.h.

  Syntax:

  ```
  void
  PARTITION_SetRealTimeMode(
              PartitionData * partition,
              bool runAsRealtime);
  ```

  Parameters:

  `partition` : Pointer to the data for this partition
  `runAsRealtime` : Indication to run execute in real time

6.3.2 Functions for Injecting Traffic from External Interfaces

The file EXATA_HOME/include/external_util.h defines some general EXata utility functions for external interfaces, including functions for sending packets at the Application Layer using UDP and TCP, creating mappings, enabling/disabling nodes, moving nodes, and changing node orientations.

- **EXTERNAL_SendDataAppLayerUDP**: This function sends a packet between two nodes using UDP. By default, the function sends the data immediately. The default destination application is APP_FORWARD. APP_FORWARD simply forwards the received data back to the external interface.

Syntax:

```
void EXTERNAL_SendDataAppLayerUDP(
     EXTERNAL_Interface *iface,
     NodeAddress from,
     NodeAddress to,
     char *data,
     int dataSize,
     clocktype timestamp = 0,
     AppType app = APP_FORWARD,
     TraceProtocolType trace = TRACE_FORWARD,
     TosType priority = IPTOS_PREC_ROUTINE,
     UInt8 ttl = TTL_NOT_SET);
```

Parameters:

`iface`	: The external interface
`from`	: The address of the sending node
`to`	: The address of the receiving node
`data`	: The data that is to be sent. If contents of the packet are not important, no actual data is sent, and this parameter is set to `NULL` and the `dataSize` parameter is used to indicate the size of the packet.
`dataSize`	: The size of the data
`timestamp`	: The time to send the packet. Pass 0 to send immediately. Defaults to 0.
`app`	: The application to send to. Defaults to APP_FORWARD.
`trace`	: The trace protocol. Defaults to TRACE_FORWARD.
`priority`	: The priority of the message
`ttl`	: The time to live

- **EXTERNAL_SendDataAppLayerTCP**: This function sends data between two nodes using TCP. The TCP connection is established the first time this function is called between a pair of nodes.By default, the function sends the data immediately. Unlike EXTERNAL_SendDataAppLayerUDP, which can deliver data to any application using UDP, EXTERNAL_SendDataAppLayerTCP delivers data only to the application APP_FORWARD. APP_FORWARD simply forwards the received data back to the external interface.

  Syntax:

```
void EXTERNAL_SendDataAppLayerTCP(
    EXTERNAL_Interface *iface,
    NodeAddress from,
    NodeAddress to,
    char *data,
    int dataSize,
    clocktype timestamp = 0,
    UInt8 ttl = TTL_NOT_SET);
```

Parameters:

`iface`	: The external interface
`from`	: The address of the sending node
`to`	: The address of the receiving node
`data`	: The data that is to be sent. If contents of the packet are not important, no actual data is sent, and this parameter is set to `NULL` and the `dataSize` parameter is used to indicate the size of the packet.
`dataSize`	: The size of the data
`timestamp`	: The time to send the packet. Pass 0 to send immediately. Defaults to 0.
`ttl`	: The time to live

- **EXTERNAL_SendDataNetworkLayer**: This function sends data between two nodes at the Network Layer. This function differs from the UDP and TCP SendData functions because it sends IP packets. The contents of the IP packet (such as a transport header or application data) must be created by the user. The packet is not passed to the interface's forward function. By default, the function sends the data immediately.

  Syntax:

```
void EXTERNAL_SendDataNetworkLayer(
    EXTERNAL_Interface* iface,
    NodeAddress from,
    NodeAddress srcAddr,
    NodeAddress destAddr,
    unsigned short identification,
    BOOL dontFragment,
    BOOL moreFragments,
    unsigned short fragmentOffset,
    TosType tos,
    unsigned char protocol,
    unsigned int ttl,
    char* payload,
    int payloadSize,
    clocktype timestamp = 0);
```

Parameters:

`iface`	: The external interface
`from`	: The address of the sending node
`srcAddr`	: The IP address of the node originally creating the packet (may be different than the `from` address)
`destAddr`	: The address of the receiving node
`identification`	: The identification field in the IP header
`dontFragment`	: Indication whether to set the *Don't Fragment* bit in the IP header
`moreFragments`	: Indication whether to set the *More Fragments* bit in the IP header
`fragmentOffset`	: The *Fragment Offset* field in the IP header
`tos`	: The *Type of Service* field in the IP header
`protocol`	: The *Protocol* field in the IP header
`ttl`	: The *Time to Live* field in the IP header
`payload`	: The data to be sent. This should include appropriate transport headers. If NULL, the payload will consist of 0's.
`payloadSize`	: The size of the data
`timestamp`	: The time to send the packet. Pass 0 to send at the interface's current time according to the interface's time function (if the interface does not have a time function, the packet is sent immediately). Defaults to 0.

- **EXTERNAL_CreateMapping**: This function creates a mapping between a key and a value. The key may be of any type and of any length, such as an IP address, a MAC address, or a generic string. The value may be anything (it is just a generic pointer). Correctly using the mapped value is the responsibility of the external interface.

  Syntax:

```
void EXTERNAL_CreateMapping(
    EXTERNAL_Interface *iface,
    char *key,
    int keySize,
    char *value,
    int valueSize);
```

  Parameters:

`iface`	: The external interface
`key`	: Pointer to the key
`keySize`	: The size of the key in bytes
`value`	: The value that the key maps to
`valueSize`	: The size of the value

- **EXTERNAL_ResolveMapping**: This function resolves a mapping created by the function EXTERNAL_CreateMapping. If a mapping for the specified key exists, the function returns 0; in this case the value associated with the key is placed in the `value` parameter and the size of the value is placed in the `valueSize` parameter. If a mapping for the specified key does not exist, the function returns non-zero and the `value` and `valueSize` parameters are invalid.

  This is an overloaded function. One of the versions of the function is described here.

Syntax:

```
int EXTERNAL_ResolveMapping(
    EXTERNAL_Interface *iface,
    char *key,
    int keySize,
    char **value,
    int *valueSize);
```

Parameters:

`iface` : The external interface
`key` : Pointer to the key
`keySize` : The size of the key in bytes
`value` : Pointer to the value (output)
`valueSize` : The size of the key in bytes (output)

Return Value:

0, if the mapping resolved; non-zero, if it did not resolve

- **EXTERNAL_ActivateNode**: This function activates a node so that it can begin processing events.

  Syntax:

```
void EXTERNAL_ActivateNode(EXTERNAL_Interface *iface, Node *node,
                           ExternalScheduleType scheduling
                                = EXTERNAL_SCHEDULE_STRICT,
                            clocktype deactivationEventTime = -1);
```

  Parameters:

  `iface` : The external interface
  `node` : The node to activate
  `scheduling` : The scheduling algorithm
  `deactivationEventTime` : The time for deactivation

- **EXTERNAL_DectivateNode**: This function deactivates a node so that it stops processing events.

  Syntax:

```
void EXTERNAL_DeactivateNode(EXTERNAL_Interface *iface,
                             Node *node,
                             ExternalScheduleType scheduling
                                  = EXTERNAL_SCHEDULE_STRICT,
                              clocktype deactivationEventTime = -1);
```

  Parameters:

  `iface` : The external interface
  `node` : The node to activate
  `scheduling` : The scheduling algorithm
  `deactivationEventTime` : The time for deactivation

- **EXTERNAL_ChangeNodePosition**: This function changes the position of a node. This function works with any coordinate system. The node orientation is not changed. The valid range of coordinate values depends on the terrain data. Coordinate values are checked to be in the proper range; if they are not in the proper range, the coordinate values are converted to values within the proper range.

Syntax:

```
void EXTERNAL_ChangeNodePosition(
       EXTERNAL_Interface *iface,
       Node *node,
       double c1,
       double c2,
       double c3,
       clocktype delay);
```

Parameters:

`iface` : The external interface
`node` : The node
`c1` : The first coordinate: latitude or x
`c2` : The second coordinate: longitude or y
`c3` : The third coordinate: altitude or z
`delay` : The delay for changing the position

- **EXTERNAL_ChangeNodeOrientation**: This function changes the orientation of a node. The node position is not changed. Azimuth/elevation values are checked to be in the proper range; if they are not in the proper range, the azimuth/elevation values are converted to values within the proper range.

  Syntax:

  ```
  void EXTERNAL_ChangeNodeOrientation(
         EXTERNAL_Interface *iface,
         Node *node,
         short azimuth,
         short elevation);
  ```

  Parameters:

  `iface` : The external interface
  `node` : The node
  `azimuth` : The azimuth, 0 <= azimuth <= 359
  `elevation` : The elevation, -180 <= elevation <= 180

- **EXTERNAL_ChangeNodePositionAndOrientation**: This function changes both the position and orientation of a node. This function works using both coordinate systems. The valid range of coordinate values depends on the terrain data. Coordinate values and azimuth/elevation values are checked to be in the proper range; if they are not in the proper range, the coordinate values and azimuth/elevation values are converted to values within the proper range.

  Syntax:

  ```
  void EXTERNAL_ChangeNodePositionAndOrientation(
         EXTERNAL_Interface *iface,
         Node *node,
         double c1,
         double c2,
         double c3,
         short azimuth,
         short elevation);
  ```

Parameters:

`iface`	: The external interface
`node`	: The node
`c1`	: The first coordinate: latitude or x
`c2`	: The second coordinate: longitude or y
`c3`	: The third coordinate: altitude or z
`azimuth`	: The azimuth, 0 <= azimuth <= 359
`elevation`	: The elevation, -180 <= elevation <= 180

- **EXTERNAL_ChangeNodePositionOrientationAndVelocityAtTime:** This function updates the position, orientation, and velocity vector of a node at a user-specified time. The velocity vector is expected to be in the same distance units used for the node's position, per one second. The speed parameter must also be provided, accurate for the provided velocity vector, and always in meters per second. Coordinate values, azimuth, elevation, and speed values are checked to be in the proper range, and are converted to the proper range if not.

  Syntax:

```
void EXTERNAL_ChangeNodePositionOrientationAndVelocityAtTime(
       EXTERNAL_Interface *iface,
       Node *node,
       clocktype mobilityEventTime,
       double c1,
       double c2,
       double c3,
       short azimuth,
       short elevation,
       double speed,
       double c1Speed,
       double c2Speed,
       double c3Speed);
```

  Parameters:

`iface`	: The external interface
`node`	: The node
`mobilityEventTime`	: The absolute simulation time (not delay) at which the mobility event should execute
`c1`	: The first coordinate: latitude or x-coordinate
`c2`	: The second coordinate: longitude or y-coordinate
`c3`	: The third coordinate: altitude or z-coordinate
`azimuth`	: The azimuth, 0 <= azimuth <= 359
`elevation`	: The elevation, -180 <= elevation <= 180
`speed`	: The speed in meters/sec
`c1Speed`	: The rate of change of the first coordinate in the units of the position, per second
`c2Speed`	: The rate of change of the second coordinate in the units of the position, per second
`c3Speed`	: The rate of change of the third coordinate in the units of the position, per second

- **EXTERNAL_ChangeNodePositionOrientationAndVelocityAtTime:** This function updates the position, orientation, and velocity vector of a node at a user-specified time. The velocity vector is expected to be in the same distance units used for the node's position, per one second. Coordinate values, azimuth, elevation, and speed values are checked to be in the proper range, and are converted to the proper range if not.

  Syntax:

```
void EXTERNAL_ChangeNodePositionOrientationAndVelocityAtTime(
       EXTERNAL_Interface *iface,
       Node *node,
       clocktype mobilityEventTime,
       double c1,
       double c2,
       double c3,
       short azimuth,
       short elevation,
       double c1Speed,
       double c2Speed,
       double c3Speed);
```

  Parameters:

`iface`	: The external interface
`node`	: The node
`mobilityEventTime`	: The absolute simulation time (not delay) at which the mobility event should execute
`c1`	: The first coordinate: latitude or x-coordinate
`c2`	: The second coordinate: longitude or y-coordinate
`c3`	: The third coordinate: altitude or z-coordinate
`azimuth`	: The azimuth, 0 <= azimuth <= 359
`elevation`	: The elevation, -180 <= elevation <= 180
`c1Speed`	: The rate of change of the first coordinate in the units of the position, per second
`c2Speed`	: The rate of change of the second coordinate in the units of the position, per second
`c3Speed`	: The rate of change of the third coordinate in the units of the position, per second

- **EXTERNAL_ChangeNodeVelocityAtTime:** This function updates the velocity vector of a node at a user-specified time. The velocity vector is expected to be in the same distance units used for the terrain coordinate system, per one second. The speed parameter must also be provided, accurate for the provided velocity vector, and always in meters per second.

  Syntax:

```
void EXTERNAL_ChangeNodeVelocityAtTime(
       EXTERNAL_Interface *iface,
       Node *node,
       clocktype mobilityEventTime,
       double speed,
       double c1Speed,
       double c2Speed,
       double c3Speed);
```

Parameters:

`iface`	: The external interface
`node`	: The node
`mobilityEventTime`	: The absolute simulation time (not delay) at which the mobility event should execute
`speed`	: The speed in m/s
`c1Speed`	: The rate of change of the first coordinate in the distance units of the terrain coordinate system, per second
`c2Speed`	: The rate of change of the second coordinate in the distance units of the terrain coordinate system, per second
`c3Speed`	: The rate of change of the third coordinate in the distance units of the terrain coordinate system, per second

- **EXTERNAL_ConfigStringPresent**: This function checks the configuration file for a string. Typically, this is used during interface registration to see if an interface name is included in the configuration file. This function only checks for the presence of a string and not the value associated with that string.

  Syntax:

  ```
  BOOL EXTERNAL_ConfigStringPresent(
         NodeInput *nodeInput,
         char *string);
  ```

  Parameters:

`nodeInput`	: The configuration file
`string`	: The string to check for

  Return Value:

  `TRUE`, if the string is present; `FALSE`, otherwise

- **EXTERNAL_ConfigStringIsYes**: This function checks the configuration file for a string and returns `TRUE` if that string is equal to "`YES`". Typically, this is used during interface registration to see if an interface name is included in the configuration file.

  Syntax:

  ```
  BOOL EXTERNAL_ConfigStringIsYes(
         NodeInput *nodeInput,
         char *string);
  ```

  Parameters:

`nodeInput`	: The configuration file
`string`	: The string to check for

  Return Value:

  `TRUE`, if the string is present and set to "`YES`"; `FALSE`, otherwise

6.3.3 Operating System-specific Utility Functions for Sockets

The file EXATA_HOME/include/external_socket.h contains functions that are useful for a socket programmer. The first set of functions are for operations on an array of variable size. The next set of functions implements host-to-network byte ordering. The final set of functions implements the socket code.

6.3.3.1 Functions for Variable-sized Array Operations

The file EXATA_HOME/include/external_socket.h defines a structure called `EXTERNAL_VarArray`, which is an array of variable size. The file external_socket.h also defines the following functions for operations on `EXTERNAL_VarArray`:

- **EXTERNAL_VarArrayInit**: This function initializes an array of type `EXTERNAL_VarArray` and allocates memory for the array.

  Syntax:

  ```
  void EXTERNAL_VarArrayInit(
         EXTERNAL_VarArray *array,
         Unsigned int size = EXTERNAL_DEFAULT_VAR_ARRAY_SIZE);
  ```

  Parameters:

  `array` : Pointer to the uninitialized array

  `size` : The initial size of the array in bytes. Defaults to EXTERNAL_DEFAULT_VAR_ARRAY_SIZE

- **EXTERNAL_VarArrayAccomodateSize**: This function increases the maximum size of an array of type `EXTERNAL_VarArray` to at least the specified size.

  Syntax:

  ```
  void EXTERNAL_VarArrayAccomodateSize(
         EXTERNAL_VarArray *array,
         unsigned int size);
  ```

  Parameters:

  `array` : Pointer to the array

  `size` : The new minimum size of the array

- **EXTERNAL_VarArrayAppendData**: This function adds data to the end of an array of type `EXTERNAL_VarArray`. The size of the array is increased if necessary.

  Syntax:

  ```
  void EXTERNAL_VarArrayAppendData(
         EXTERNAL_VarArray *array,
         char *data,
         unsigned int size);
  ```

  Parameters:

  `array` : Pointer to the array

  `data` : Pointer to the data to add

  `size` : The size of the data to add

- **EXTERNAL_VarArrayConcatString**: This function adds a string to the end of an array of type `EXTERNAL_VarArray`, including the terminating `NULL` character. This function assumes that the previous data in the array is also a string, i.e., several bytes of data terminated with a `NULL` character. If this is not the case, then the function EXTERNAL_VarArrayAppendData should be used instead.

  Syntax:

  ```
  void EXTERNAL_VarArrayConcatString(
         EXTERNAL_VarArray *array,
         char *string);
  ```

Parameters:

`array` : Pointer to the array

`string` : The string

- **EXTERNAL_VarArrayFree**: This function frees all memory allocated to an array of type `EXTERNAL_VarArray`. This function must be called once the use of the array is over.

  Syntax:

  ```
  void EXTERNAL_VarArrayFree(EXTERNAL_VarArray *array);
  ```

  Parameters:

  `array` : Pointer to the array

6.3.3.2 Host-to-Network Byte Order Functions

These functions, implemented in the file EXATA_HOME/include/external_socket.h, reverse the byte order of variables from host to network order and vice versa.

- **EXTERNAL_hton**: This function converts data from host byte order to network byte order.

  Syntax:

  ```
  void EXTERNAL_hton(void* ptr, unsigned int size);
  ```

  Parameters:

  `ptr` : Pointer to the data

  `size` : Size of the data

- **EXTERNAL_ntoh**: This function converts data from network byte order to host byte order

  Syntax:

  ```
  void EXTERNAL_ntoh(void* ptr, unsigned int size);
  ```

  Parameters:

  `ptr` : Pointer to the data

  `size` : Size of the data

6.3.3.3 External Socket Functions

The external socket functions implement a socket. These functions are implemented in the file EXATA_HOME/include/external_socket.h.

- **EXTERNAL_SocketInit**: This function initializes a socket. This function must be called before any other socket API calls on the individual socket.

  Syntax:

  ```
  void EXTERNAL_SocketInit(EXTERNAL_Socket *s,
                           BOOL blocking = TRUE,
                           BOOL threaded = FALSE);
  ```

  Parameters:

  `s` : Pointer to the socket

  `blocking` : Flag that indicates whether blocking is enabled

  `threaded` : Flag that indicates whether multithreading is enabled

- **EXTERNAL_SocketValid**: This function checks if a socket connection is valid, i.e., the socket is open and no errors have occurred.

  Syntax:

  BOOL EXTERNAL_SocketValid(EXTERNAL_Socket *socket);

  Parameters:

  `s` : Pointer to the socket

  Return Value:

  `TRUE`, if the socket is valid; `FALSE`, if invalid

- **EXTERNAL_SocketListen**: This function listens for connections on a socket. It accepts a socket structure, `listenSocket`, and a port number, `port`, as parameters. `listenSocket` is used to listen for an incoming connection. If a successful socket connection is created, the socket parameter `connectSocket` is assigned the newly created socket connection, which is set to non-blocking mode. If `listenSocket` has already been initialized by an earlier call to EXTERNAL_SocketListen, the `port` parameter is ignored.

  Syntax:

```
EXTERNAL_SocketErrorType EXTERNAL_SocketListen(
       EXTERNAL_Socket *listenSocket,
       int port,
       EXTERNAL_Socket *connectSocket);
```

  Parameters:

  `listenSocket` : Pointer to the socket to listen on

  `port` : The port to listen on

  `connectSocket` : Pointer to the socket that receives the established connection

  Return Value:

  `EXTERNAL_NoError`, if successful; an error indication, if not successful

- **EXTERNAL_SocketConnect**: This function connects to a listening socket. The socket is set to non-blocking mode.

  Syntax:

```
EXTERNAL_SocketErrorType EXTERNAL_SocketConnect(
       EXTERNAL_Socket *socket,
       char *address,
       int port,
       int maxAttempts);
```

  Parameters:

  `socket` : Pointer to the socket

  `address` : String representation of the address to connect to

  `port` : The port to connect to

  `maxAttempts` : Number of times to attempt connecting before an error is returned

  Return Value:

  `EXTERNAL_NoError`, if successful; an error indication, if not successful

- **EXTERNAL_SocketSend**: This function sends data on a connected socket. It is possible that the send would result in a block: If the `block` parameter is `FALSE`, then `EXTERNAL_DataNotSent` is returned, and no data is sent. If the `block` parameter is `TRUE`, then the function blocks until the data can be sent.

Syntax:

```
EXTERNAL_SocketErrorType EXTERNAL_SocketSend(
        EXTERNAL_Socket *s,
        char *data,
        unsigned int size,
        BOOL block = TRUE);
```

Parameters:

`socket` : Pointer to the socket

`data` : Pointer to the data

`size` : Size of the data

`block` : Indication whether this call may block. Defaults to `TRUE`.

Return Value:

`EXTERNAL_NoError`, if successful; an error indication, if not successful

- **EXTERNAL_SocketSend**: This is a wrapper for the above overloaded function.

Syntax:

```
EXTERNAL_SocketErrorType EXTERNAL_SocketSend(
        EXTERNAL_Socket *s,
        EXTERNAL_VarArray *data,
        BOOL block = TRUE);
```

Parameters:

`socket` : Pointer to the socket

`data` : Pointer to the array to send

`block` : Indication whether this call may block. Defaults to `TRUE`.

Return Value:

`EXTERNAL_NoError`, if successful; an error indication, if not successful

- **EXTERNAL_SocketRecv**: This function receives data on a connected socket. It is possible that the send would result in a block: If the `block` parameter is `FALSE`, the `receiveSize` parameter is set to the amount of data received before the block. This amount could be any value between 0 and `size` - 1.

Syntax:

```
EXTERNAL_SocketErrorType EXTERNAL_SocketRecv(
        EXTERNAL_Socket *s,
        char *data,
        unsigned int size,
        unsigned int *receiveSize,
        BOOL block = TRUE);
```

Parameters:

`socket` : Pointer to the socket

`data` : Pointer to the destination

`size` : The amount of data to receive in bytes

`receiveSize` : The number of bytes received. This could be less than the specified size if an error occurred or if the `block` parameter is `FALSE`.

`block` : `TRUE` if the call can block, `FALSE` if non-blocking. Defaults to `TRUE`.

Return Value:

`EXTERNAL_NoError`, if successful; an error indication, if not successful

- **EXTERNAL_SocketClose**: This function closes a socket. This function must be called for each socket that is listening or connected.

  Syntax:

  ```
  EXTERNAL_SocketErrorType EXTERNAL_SocketClose(EXTERNAL_Socket *s);
  ```

  Parameters:

  `s` : Pointer to the socket

  Return Value:

  `EXTERNAL_NoError`, if successful; an error indication, if not successful

7 Dynamic API

The dynamic API allows users and programs to dynamically modify and monitor a EXata simulation. Using the dynamic API, a user or program can change values of variables and can be notified when statistics change during execution. The dynamic API operates between the external interface API and the simulation. The external interface API is responsible for communicating with the external program and sending commands from the external program to the dynamic API. The dynamic API interacts with the EXata simulation, giving results of commands back to the external interface API, which in turn sends results back to the external program. This modular design allows different front-ends to use the dynamic API. Therefore each external program can use the most appropriate interface. Figure 7-1 illustrates the dynamic API's relation to EXata.

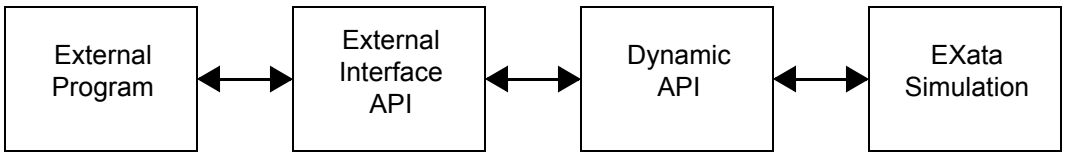


FIGURE 7-1. Dynamic API

Section 7.1 describes the implementation of the dynamic API in EXata. Section 7.2 describes how the dynamic API is used by an external program. Section 7.3 describes how to enable a protocol for dynamic manipulation. Section 7.4 describes how to define new dynamic data types.

7.1 Implementation of the Dynamic API

This section describes the implementation of the dynamic API in EXata.

7.1.1 Dynamic Objects

The dynamic API uses the object oriented features of C++. The dynamic API is implemented by means of dynamic objects which are derived from the base class `D_Object`. The `D_Object` class is defined in EXATA_HOME/include/dynamic.h and has the following characteristics:

- Type: Variable, statistic or command
- Level: The object's location in the hierarchy (see Section 7.1.3)
- Permissions: Readable, writable, executable
- Listeners: If listening is enabled (see Section 7.1.4), an array of all listeners attached to this object
- Action functions: Functions to read a variable's value, modify a variable's value, or execute a command. Not all data types implement all action functions.

7.1.2 Built-in Dynamic Objects

The following dynamic objects are pre-defined in EXata. These objects are defined in EXATA_HOME/include/dynamic_vars.h.

- `D_Int32`: A 4-byte integer
- `D_UInt32`: A 4-byte unsigned integer
- `D_Int64`: An 8-byte signed integer
- `D_Float32`: A-4 byte floating point number
- `D_Float64`: An 8-byte floating point number
- `D_NodeAddress`: A node address or node identifier
- `D_String`: A string
- `D_Clocktype`: A clocktype, 8-byte integer

7.1.3 Hierarchy of Objects

The dynamic objects are organized in a hierarchy structure. This hierarchy structure is similar to the directory structure used by a computer's operating system. All dynamic objects are organized in the hierarchy based on levels which form a path. For instance, the dynamic object corresponding to the AODV variable `numRequestsInitiated` for node 129 and interface address 192.168.0.129 is located at the following path: `/node/129/interface/192.168.0.129/aodv/numRequestsInitiated`.

The hierarchy is implemented using the `D_Hierarchy` class defined in EXATA_HOME/include/dynamic.h.

7.1.4 Listening

A dynamic object may have one or more listeners attached to it. A listener is a function that is automatically notified when the value of the data component of the dynamic object (see Section 7.1.5) changes. The listener can implement a filter for the updates, thereby determining the thresholds for notification.

Note: By default, listening is enabled for all variables. Listening can be disabled upon customer request, but this would require a customized distribution of EXata.

7.1.5 Data Component of a Dynamic Object

Some dynamic objects, such as variables and statistics, have a data component. The data component of an object and the dynamic object are internally treated as two separate entities. The data component is contained within the protocol's data structure. For example, the AODV variable `numRequestsInitiated`, discussed in Section 7.1.3, would have a data component that is an integer. This integer contains the value of the dynamic variable and is part of the AODV data structure. The object (including the path `/node/129/interface/192.168.0.129`) is part of the dynamic hierarchy.

If listening is disabled the data component is a simple variable. For example, if listening is disabled, the built-in dynamic data type `D_Int32` is equivalent to an `Int32`. The dynamic data type `D_Int32` is defined in EXATA_HOME/include/dynamic_vars.h. If listening is enabled the data component is a class wrapper for the simple variable. The class wrapper signals the dynamic API when the variable's value is modified. The dynamic object is located in the hierarchy (see Section 7.1.3) and contains a pointer to the data component.

Figure 7-2 illustrates the relationship between a dynamic object and its data component. The CBR protocol has a dynamically enabled variable `interval` of type `D_Clocktype`. This variable is identified with an un-named dynamic object of type `D_ClocktypeObj`. The `D_ClockTypeObj` object acts as an intermediary between the `clocktype` data and the dynamic API hierarchy. The `D_ClocktypeObj` is added to the hierarchy at the appropriate position, in this case under …`/cbr/1024/interval`. The `1024` level indicates that the CBR source port is 1024.

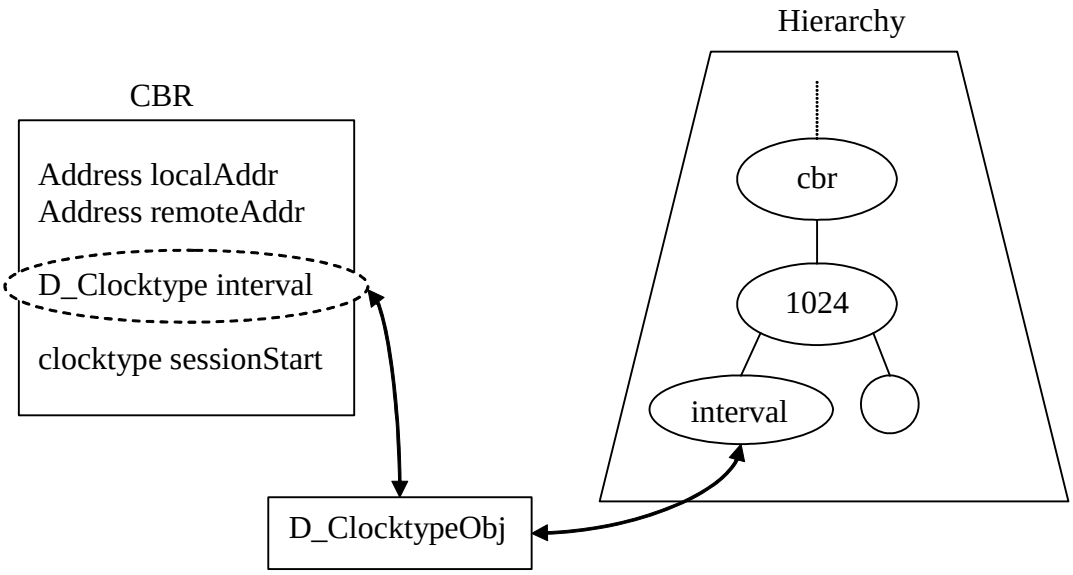


FIGURE 7-2. Data Components and Hierarchy of Objects

7.1.6 Dynamic Commands

Dynamic commands are similar in implementation to dynamic data types except that a dynamic command does not have a built-in data component. For this reason it is useful to define member variables within the command.

Dynamic commands are derived from the base class `D_Command`, which is defined in EXATA_HOME/include/dynamic.h. As an example, Figure 7-3 shows the declaration of the dynamic command object `D_QshChangeModeCommand` in the qsh addon file EXATA_HOME/interfaces/qsh/src/qsh_interface.h. This command is used for changing qsh's execution mode between real-time, time-managed, and none.

The `D_QshChangeModeCommand` object records which interface it belongs to in the variable `iface` and sets the interface using the constructor. The function ExecuteAsString is the action function for this object. Function ExecuteAsString, shown in Figure 7-4, reads the incoming parameters passed as the input parameter `in`, changes qsh's execution mode, and sends back results in the output parameter `out`. Function ExecuteAsString of the object `D_QshChangeModeCommand` is implemented in EXATA_HOME/interfaces/qsh/src/qsh_interface.cpp.

```
class D_QshChangeModeCommand : public D_Command
{
    private:
        EXTERNAL_Interface* iface;

    public:
        D_QshChangeModeCommand(EXTERNAL_Interface* newIface)
        {
            iface = newIface;
        }

        virtual void ExecuteAsString(const std::string& in, std::string& out);
};
```

FIGURE 7-3. Example Dynamic Command

```
void D_QshChangeModeCommand::ExecuteAsString(
    const std::string& in,
    std::string& out)
{
    QshData* data = (QshData*) iface->data;
    D_Hierarchy* h = &iface->partition->dynamicHierarchy;

    if (in == "real-time")
    {
        ...
    }
    else if (in == "time-managed")
    {
       ...
    }
    else if (in == "none")
    {
        ...
    else
    {
        out = std::string("FAILURE -- Unknown mode \") + in +
              "\" (must be \"real-time\", \"time-managed\" or \"none\")");
    }
}
```

FIGURE 7-4. Action Function of a Dynamic Command

7.2 Using the Dynamic API from an External Interface

An external interface may access the dynamic hierarchy through the partition structure. An external interface uses the dynamic API through the following functions of the `D_Hierarchy` class, which is defined in EXATA_HOME/include/dynamic.h:

- ReadAsString: This function reads the value of the object with the given path. The function throws an exception if the path does not exist or if the path does not resolve to an object. that is readable.
- WriteAsString: This function writes the value to the object with the given path. The function throws an exception if the path does not exist or if the path does not resolve to an object. that is writable.
- ExecuteAsString: This function executes the object with the given path. The function throws an exception if the path does not exist or if the path does not resolve to an object. that is executable.
- AddListener: This function adds a new listener to the object with the given path. The function throws an exception if the path does not resolve to an object. The function is defined only if listening is enabled.
- RemoveListeners: This function removes all listeners from an object and frees their memory. The function is defined only if listening is enabled.

A typical external interface uses the dynamic API by calling functions ReadAsString, WriteAsString and ExecuteAsString. Figure 7-5 shows a sample code segment that makes a call to the function ReadAsString. In this example, the interface reads the value of `node/1/cbr/1024/numPktsSent` and prints the result. If the path is invalid or the object cannot be read, an exception is thrown, causing an error message to be printed.

```
...
std::string value;
std::string errString;
try
{
    interface->partition->dynamicHierarchy.ReadAsString(
                                        "node/1/cbr/1024/numPktsSent", value);
    printf("The value is \"%s\"\n", value.c str());
}
catch (D_Exception &e)
{
    e.GetFullErrorString(errString);
    printf("Error: \"%s\"\n", errString.c str());
}
...
```

FIGURE 7-5. Example Code to Use Function ReadAsString

Functions WriteAsString and ExecuteAsString work in a similar way.

If listening is enabled, the AddListener and RemoveListeners member functions of the `D_Hierarchy` class are also available. New `D_Listener` and `D_ListenerCallback` objects should be created for each listener that is added. Figure 7-6 shows a sample code segment that makes a call to the function AddListener. In this example, a `ListenerCallback` class is defined. It overloads the () operator. In the second part of the code segment, a listener is added to the object `node/1/cbr/1024/numPktsSent`. The listener is `D_ListenerPercent`, which invokes the callback function whenever the variable changes by 10% compared to the last time the callback function was invoked. A sample string printed by the callback function is: "`The value of node/1/cbr/1024/numPktsSent changed to 350`".

```
...
class ListenerCallback : public D_ListenerCallback
{
    public:
        virtual void operator () (const std::string& newValue)
        {
            printf("The variable changed to \"%s\"\n", newValue.c str());
        }
};
...
std::string errString;
try
{
    interface->partition->dynamicHierarchy.AddListener(
        "node/1/cbr/1024/numPktsSent",       // path
        const std::string& listenerType,
        "0.10",                              // arguments
        "qsh"                                // listener tag
        new QshListenerCallback());
}
catch (D_Exception &e)
{
    e.GetFullErrorString(errString);
    printf("Error: \"%s\"\n", errString.c str());
}
```

FIGURE 7-6. Example Code to Use Function AddListener

The external interface may remove listeners by calling function RemoveListeners of `D_Hierarchy` class. Figure 7-7 shows a sample code segment that makes a call to the function RemoveListeners. In this example, all listeners of the object `node/1/cbr/1024/numPktsSent` are removed.

```
std::string errString;
try
{
    interface->partition->dynamicHierarchy.RemoveListeners(
        "node/1/cbr/1024/numPktsSent",
        "qsh");    // listener tag
}
catch (D_Exception &e)
{
    e.GetFullErrorString(errString, "qsh");
    printf("Error: \"%s\"\n", errString.c str ());
}
```

FIGURE 7-7. Sample Code to Use Function RemoveListeners

7.3 Dynamically Enabling a Protocol

This section describes how to enable a dynamic variable and mentions issues to be aware of. Keep in mind that a dynamic variable becomes an object when listening is enabled.

To dynamically enable a protocol, do the following:

1. Change the type of the variables to the corresponding dynamic data types (see Section 7.3.1).
2. Add a dynamic object in the hierarchy for each dynamic variable (see Section 7.3.2).
3. Set the permissions (readable, writable, executable) of the dynamic objects (see Section 7.3.3).
4. Allocate memory for the dynamically enabled protocol data structure (see Section 7.3.4).

7.3.1 Declare Dynamic Variables

To make a protocol variable dynamically accessible, change the type of the variable to the corresponding dynamic data type. The built-in dynamic data types of EXata are listed in Section 7.1.2. For example, in the CBR client data structure `AppDataCbrClient` shown below, variables `interval` and `numBytesSent` are dynamically enabled and the rest of the variables are not dynamically enabled. `AppDataCbrClient` is defined in EXATA_HOME/libraries/developer/src/app_cbr.h.

```
typedef struct struct_app_cbr_client_str
{
    Address localAddr;
    Address remoteAddr;
    D_Clocktype interval;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastSent;
    clocktype endTime;
    BOOL sessionIsClosed;
    D_Int64 numBytesSent;
    UInt32 numPktsSent;
    UInt32 itemsToSend;
    UInt32 itemSize;
    short sourcePort;
    Int32 seqNo;
    D_UInt32 tos;
}AppDataCbrClient;
```

7.3.2 Adding a Dynamic Object to the Hierarchy

For each dynamically enabled variable, add a dynamic object to the hierarchy by doing the following:

1. Create the path for the object.
2. Add the object at the appropriate path.

A path is created by calling an appropriate member function of class `D_Hierarchy`, which is defined in EXATA_HOME/include/dynamic.h. The functions used to create a path at different levels are:

- CreatePartitionPath: This function creates a path at the partition level.
- CreateNodePath: This function creates a path at the node level.

- CreateExternalInterfacePath: This function creates a path at the external interface level.
- CreateApplicationPath: This function creates a path at the application level.
- CreateTransportPath: This function creates a path at the transport level.
- CreateNetworkPath: This function creates a path at the network level.
- CreateRoutingPath: This function creates a path at the routing level.
- CreateMacPath: This function creates a path at the MAC level.
- CreatePhynPath: This function creates a path at the PHY level.

Each function takes arguments necessary to determine the path. If an object can be added at that path the function returns TRUE. An object can not be added at a given path if the path already exists or if the protocol is not configured to be dynamically enabled.

After the path has been created the object can be added by calling the `D_Hierarchy` function AddObject. As an example, consider the CBR function AppCbrClientNewCbrClient shown in Figure 7-8. Function AppCbrClientNewCbrClient first creates a path for the object `interval` by calling function CreateApplicationPath. If given permission by the hierarchy, i.e., if CreateApplicationPath returns TRUE, AppCbrClientNewCbrClient adds an object to the newly created path by calling the function AddObject. Note that a new object of class `D_ClocktypeObj` is added to the hierarchy that references the variable `cbrClient->interval`. Similarly, a path is created for the object `numBytesSent` and a new object of class `D_Int64Obj` is added to the hierarchy that references the variable `cbrClient->numBytesSent`.

Function AppCbrClientNewCbrClient is implemented in EXATA_HOME/libraries/developer/src/app_cbr.cpp. `D_ClocktypeObj`, `D_Int64Obj` and the other classes implementing dynamic objects are defined in EXATA_HOME/include/dynamic_vars.h.

```
AppDataCbrClient *
AppCbrClientNewCbrClient(Node *node,
                         Address localAddr,
                         Address remoteAddr,
                         Int32 itemsToSend,
                         Int32 itemSize,
                         clocktype interval,
                         clocktype startTime,
                         clocktype endTime,
                         TosType tos)
{
    AppDataCbrClient *cbrClient;
    cbrClient = (AppDataCbrClient *) MEM_malloc(sizeof(AppDataCbrClient));
    memset(cbrClient, 0, sizeof(AppDataCbrClient));
    /*
     * fill in cbr info.
     */
    memcpy(&(cbrClient->localAddr), &localAddr, sizeof(Address));
    memcpy(&(cbrClient->remoteAddr), &remoteAddr, sizeof(Address));
    cbrClient->interval = interval;
    ...
    // Add CBR variables to hierarchy

    std::string path;
    D_Hierarchy *h = &node->partitionData->dynamicHierarchy;

    if (h->CreateApplicationPath(
            node,                   // node
            "cbrClient",            // protocol name
            cbrClient->sourcePort,  // port
            "interval",             // object name
            path))                  // path (output)
    {
        h->AddObject(
            path,
            new D_ClocktypeObj(&cbrClient->interval));
    }
    if (h->CreateApplicationPath(
            node,
            "cbrClient",
            cbrClient->sourcePort,
            "numBytesSent",
            path))
    {
        h->AddObject(
            path,
            new D_Int64Obj(&cbrClient->numBytesSent));
    }
   ...
}
```

FIGURE 7-8. Adding Objects to the Hierarchy

Pitfalls to Avoid

- Dynamic variables are C++ objects if listening is enabled (by default, listening is enabled). Several complications can arise from this:
  - The `struct`/`class` containing the variable should be allocated using a new call instead of MEM_malloc. This allows the dynamic object to be instantiated.
  - Dynamic objects can no longer be passed directly to the printf family of functions. Instead, they should be cast to their underlying data type. For example, a `D_Int32` variable should be cast to an `Int32` before printing.
  - Always test the protocol with listening enabled and disabled.
- Dynamic variables may change at any time. Before dynamically enabling a variable make sure that a changing variable will not interfere with the protocol. If not, set the variable as not writable (see Section 7.3.3).

7.3.3 Object Permissions

Objects may be read, written and executed. Object permissions are set automatically when the object is created. Variables may be read and written. Commands may be executed.

Objects may change their permissions by calling the `D_Hierarchy` functions SetReadable, SetWriteable and SetExecutable. As an example, the following code segment creates a path for the object `numNodes`, adds an object of class `D_Int32Obj` at the newly created path, and sets the permission for the newly created object to be non-writable.

```
if (h->CreatePartitionPath(
        partitionData,
        "numNodes",
        path))
{
   h->AddObject(
        path,
        new D_Int32Obj(&partitionData->numNodes));
   h->SetWriteable(
        path,
        FALSE);
}
```

7.3.4 Initializing a Dynamically Enabled Protocol

A dynamically enabled protocol that is allocated using MEM_malloc must have the entire data structure set to 0 using the function memset. It is not enough to set each dynamic object to 0. This is necessary because there is extra data for each dynamic object that must be initialized.

If the dynamic protocol is allocated using the C++ memory management function new then the data structure may be set to 0 using the function, memset although it is not required.

7.3.5 Dynamic Strings

If listening is enabled, i.e., D_LISTENING_ENABLED is defined, then dynamic objects cannot be passed directly to the function printf. They must be cast to the appropriate type. As an example, consider printing a D_Int32 object aodv->numRequestsInitiated. The value of this object can be printed by using the following line:

```
printf("numRequestsInitiated = %d\n", (int) aodv->numRequestsInitiated);
```

7.4 Defining New Dynamic Data Types

New dynamic data types may be defined by the user. This is accomplished by creating a new class that inherits from the most appropriate base class. Defining a new data type comprises two steps: defining the data component (see Section 7.4.1) and defining the object component (see Section 7.4.2). We illustrate these steps by studying the implementation of the dynamic data type D_Int32.

7.4.1 Defining the Data Component

Figure 7-9 shows the definition of the data component of D_Int32. If listening is disabled, i.e., D_LISTENING_ENABLED is not defined, then D_Int32 is equivalent to the type int. If listening is enabled, then D_Int32 is derived from the class D_SimpleObject. D_SimpleObject is a small class that is used to link the data of an object to the object that is stored in the hierarchy. D_SimpleObject implements the changed function that notifies the dynamic API when the object's value changes.

D_Int32 and D_SimpleObject are declared in EXATA_HOME/include/dynamic_vars.h.

```
ifdef D_LISTENING_ENABLED
class D_Int32 : public D_SimpleObject
{
    private:
        Int32 value;

    public:
        Int32& operator = (Int32 newValue)
        {
            value = newValue;
            Changed();
            return value;
        }

        Int32& operator ++(Int32)
        {
            value++;
            Changed();
            return value;
        }
        operator Int32()
        {
            return value;
        }
        operator Int32() const
        {
            return value;
        }
};
#else // D_LISTENING_ENABLED
typedef Int32 D_Int32;
#endif // D_LISTENING_ENABLED
```

FIGURE 7-9. Data Component of `D_Int32`

7.4.2 Defining the Object Component

Figure 7-10 shows the definition of the object component corresponding to `D_Int32`, which is called `D_Int32Obj`. The object contains a pointer to the data component that is named `value`. The constructor function links this object to the data component. If listening is enabled, the data component is linked back to this object. The functions IsNumeric and GetDouble over-ride virtual `D_Variable` functions. Function IsNumeric states that this variable is numeric and function GetDouble provides a numeric value for this variable. The final two functions ReadAsString and WriteAsString also over-ride virtual `D_Object` functions. They are the *action functions* for this object that determine this object's behavior. In this case they simply convert the object's value to and from a string.

`D_Int32Obj` is declared in EXATA_HOME/include/dynamic_vars.h. `D_Variable` and `D_Object` are declared in EXATA_HOME/include/dynamic.h.

```
class D_Int32Obj : public D_Variable
{
    private:
        D_Int32* value;

    public:
        D_Int32Obj(D_Int32* newValue)
        {
            value = newValue;
#ifdef D_LISTENING_ENABLED
            value->SetObject(this);
#endif // D_LISTENING_ENABLED
        }

        virtual BOOL IsNumeric()
        {
            return TRUE;
        }
        virtual double GetDouble()
        {
            return (double) *value;
        }

        void ReadAsString(std::string& out)
        {
            std::ostringstream oss;
            oss << (Int32) *value;
            out = oss.str();
        }

        void WriteAsString(const std::string& in)
        {

            std::istringstream iss(in);
            Int32 intVal;
            iss >> intVal;
            *value = intVal;
        }
};
```

FIGURE 7-10. Object Component `D_Int32Obj`

A Coding Guidelines for 64-bit Platforms

This appendix provides coding guidelines for developing EXata models for 64-bit platforms. It also discusses some compatibility issues when converting models between 32-bit and 64-bit platforms.

A.1 Introduction

The major differences between the 32-bit and the 64-bit EXata development environments arise from the fact that the 32-bit EXata implementation is based on the ILP32 data model, whereas the 64-bit EXata implementation is based on the LP64 data model (for Linux systems) or the P64 data model (for Windows systems). In the ILP32 data model, `long` and pointer types are 32 bits long, where as in the LP64 and P64 data models, `long` and pointer types are 64 bits long. The other fundamental data types have the same length in the 32-bit and 64-bit data models.

Table A-1 lists the size of common data types in the three data models.

TABLE A-1. Size of Common Data Types

Type	ILP32 Size (bytes)	LP64 Size (bytes)	P64 Size (bytes)
`int`	4	4	4
`long`	4	8	8
Pointers	4	8	8
`size_t`	4	8	8
`time_t`	4	8	8

A.2 Coding Guidelines and Compatibility Issues

This section gives some coding guidelines and lists some compatibility issues for developing EXata models for 64-bit platforms.

1. Use data types defined in EXata.

EXata has defined some data types which have the same size on all platforms. Use these data types in order to avoid problems arising from data types having different sizes on different platforms. These data types are defined in EXATA_HOME/include/types.h and are listed in Table A-2.

TABLE A-2. Fixed-size Data Types Defined in EXata

Data Type	Size
`Int8`	8-bit integer
`UInt8`	8-bit unsigned integer
`Int16`	16-bit integer
`UInt16`	16-bit unsigned integer
`Int32`	32-bit integer
`UInt32`	32-bit unsigned integer
`Int64`	64-bit integer
`UInt64`	64-bit unsigned integer
`Float32`	32-bit float
`Float64`	64-bit float

Polymorphic Data Type `IntPtr`

In addition to the above fixed-size data types, EXata also defines the polymorphic type `IntPtr` for pointer arithmetic. `IntPtr` is 32 bits long on 32-bit platforms and 64 bits long on 64-bit platforms.

2. Use `Int32` or `Int64` instead of `long`.

The data types `long` and `unsigned long` have different lengths on 32-bit and 64-bit platforms. Avoid using these data types. Use `Int32` (or `int`) and `UInt32` (or `unsigned int`) for signed and unsigned 32-bit integers, respectively. Use `Int64` and `UInt64` for signed and unsigned 64-bit integers, respectively.

3. Avoid assignment among variables of different sizes.

Truncation problems can arise when assignments are made between 64-bit and 32-bit data items. Since `int`, `long`, and pointer data types are all 32 bits long in the ILP32, mixed assignments between these data types do not present any special problems. However, in the LP64 and P64 data models, these data types have different sizes and assignment among them may result in truncation errors, for instance, when a pointer variable is assigned to an `int` variable.

To avoid problems arising from different lengths of data types:

- Do not use `int` and `long` types interchangeably, since this may lead to truncation of significant digits or unexpected results.
- Do not pass an argument of type `long` or a pointer to functions expecting a type `int` argument, as this may lead to truncation.

- Do not exchange pointers and `int` types, since this may cause segmentation faults.
- Do not assign a `long` type to a `float`, since this may cause loss of accuracy.
- Do not cast pointers to `int` or `unsigned int`, since they may have unequal sizes on some platforms.

Note **Keep in mind that the sizeof() function returns an `unsigned long`. Do not pass sizeof() to a function expecting an argument of type `int` or assign or cast it to an `int` since this may cause truncation and loss of data.**

4. Be aware of changes in `struct` size because of different paddings.

   Compilers introduce padding in structures in order to align the structure members. Because some data types have different lengths on 32-bit and 64-bit platforms, the size of the padding introduced to align structure members may be different on 32-bit and 64-bit platforms.

   Consider the following example of a `struct` definition:

```
struct myStruct
{
   long myLong;
   char myChar;
   struct myStruct* left;
   short myShort;
   struct myStruct* right;
}
```

   In this example, the same structure definition has different sizes and the structure members have different offsets in different data models. In the ILP32 data model, this data structure contains 20 bytes, as shown in Figure A-1.

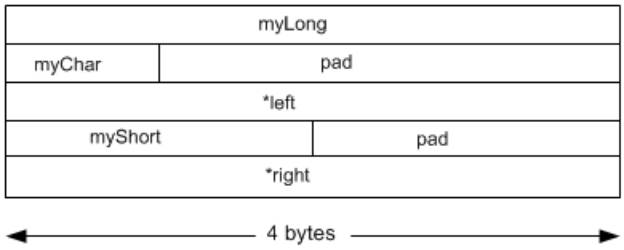


FIGURE A-1. Structure Alignment in ILP32 Data Model

   In the LP64 data model, this data structure contains 40 bytes, as shown in Figure A-2.

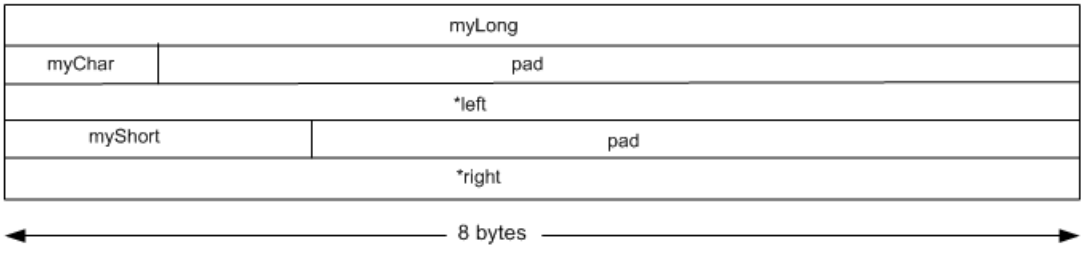


FIGURE A-2. Structure Alignment in LP64 Data Model

5. Be aware of alignment effects of bit fields.

   In the LP64 data model, unqualified bit fields are unsigned by default. In the ILP32 data model, unqualified bit fields are signed by default. Bit fields of enumerated types are signed if the enumeration base type is signed and unsigned if the enumeration base type is unsigned.

   In the LP64 data model, unnamed, non-zero length bit fields do not affect the alignment of a structure or union. In the ILP32 data model, unnamed, non-zero length bit fields affect the alignment of structures and unions.

   Also keep in mind that if you use `long` bit fields in 64-bit mode, their exact alignment may change in future versions of the compiler, even if the bit field is less than 32 bits in length.

6. The data model determines whether enumerated types are signed or unsigned.

   In the LP64 data model, enumerated types are signed only if one or more of the enumeration constants defined for that type is negative. If all enumeration constants are non-negative, the type is unsigned.

   In the ILP32 data model, enumerated types are always signed.

7. Be aware of data type used for intermediate results of operations.

   Consider the operation:

   a = b operation c

   The data type used for the intermediate results of the operation depends on the types of *b* and *c*. The intermediate result is then promoted to the type of *a*. Assuming that the intermediate result has the same data type as the type of *a* may result in erroneous or unexpected results. This is particularly true when dealing with bit shifts and bit masks because programmers often assume that the operations are performed in variables that have the same data type as the result.

   In the above example, if *c* is a 64-bit data type, but *b* and *c* are 32-bit data types, then a 32-bit data type is used for the intermediate result. This may result in an overflow or truncation before the intermediate result is assigned to *a*.

   In the following example, the left operand, 1, is a numeric constant, which the compiler treats as a 32-bit value in both ILP32 and LP64 data models:

   ```
   UInt64 y;
   y = (1 << 32);
   ```

   The bit shift operation results in an overflow in both ILP32 and LP64 data models because a 32-bit data type is used for the intermediate result in both data models. In 64-bit mode, the final result is undefined.

   Use suffixes such as `L` and `UL` for long and unsigned long if you need long constants. For example, in 64-bit mode, the above code fragment can be changed to:

   ```
   UInt64 y;
   y = (1L << 32);
   ```

   In 64-bit mode, `y` is assigned 2^32 as a result of the above operation.

8. Use platform-independent formats for printing and reading 64-bit values.

   There is a special printf/scanf formatting string for 64-bit values. However, Windows and Linux platforms define different formatting strings: `"I64"` in Windows and `"ll"` in Linux. EXata defines a platform-independent formatting string, `TYPES_64BITFMT`, for 64-bit values. It is recommended that you use `TYPES_64BITFMT` instead of `"I64"` or `"ll"`.

Example:

```
printf ("%I64d", clock);                 // AVOID!
printf ("%lld", clock);                  // AVOID!

printf ("%" TYPES_64BITFMT "d", clock); // Correct
```

9. Use data declaration macros for 64-bit immediate values.

Use data declaration macros for defining a 64-bit immediate value. EXata provides the following platform-independent macros for declaring 64-bit immediate values:

- TYPES_ToInt64(n)
- TYPES_ToUInt64(n)
- TYPES_ToIntPtr(n)

Example:
The following two declarations declare the same 64-bit constant. However, the first definition is valid only on Linux platforms while the second declaration is valid on all platforms.

```
#define CLOCKTYPE_MAX 0x7fffffffffffffffLL              //AVOID!
#define CLOCKTYPE_MAX TYPES_ToInt64(0x7fffffffffffffff) //Correct
```

10. Be aware that the default return type of a function is `int`.

By default, a function without an explicit return type returns an `int`. On 64-bit platforms, the `int` and pointer types have different sizes. Therefore, any function that returns a pointer should be declared with an explicit return type when compiling in 64-bit mode. Otherwise, the compiler will assume the function returns an `int` and truncate the resulting pointer, even if the returned value is assigned to a valid pointer.

Example:
Function calloc is defined in malloc.h with an implicit return type. The following assignment works on 32-bit platforms (because int and pointer types have the same size) but will cause an error on 64-bit platforms because the pointer returned by calloc is truncated.

```
a =(char *) calloc(25);
```

Even the type casting does not avoid this problem because the pointer returned by calloc is truncated before the type casting operation.

A.3 References

The following sites provide further guidelines on coding for 64-bit platforms:

- http://developers.sun.com/prodtech/solaris/reference/techart/index.html
- http://docs.hp.com/en/5966-9844/

B Coding Guidelines for Multi-Processor Platforms

This appendix provides coding guidelines for developing EXata models that run correctly on a multiprocessor (parallel) architecture.

Section B.1 gives some general guidelines for developing parallel-safe models. Section B.2 lists some issues that arise when developing parallel-safe models that are intended to work with external interfaces.

B.1 General Guidelines

The following rules should be followed when developing parallel-safe models:

1. Do not use global variables (see Section B.1.1).
2. Nodes should not access other nodes (see Section B.1.2).
3. Provide lookahead in MAC protocols (see Section B.1.3).
4. Do not violate inter-layer APIs (see Section B.1.4).

B.1.1 Global Variables

Do not use static or global variables for the following reasons.

1. Data in a parallel simulation must be accessed in time sequence, but since shared variables are outside the control of the simulation engine, time sequencing cannot be maintained even if the data is protected by mutex locks. (Independent of timing issues, locking of data via mutex locks or semaphores is slow and effectively makes the parallel processes sequential.)
2. It is usually safe to use global data that are written once (during initialization) and are subsequently read-only, provided that the initialization is properly sequenced and the same data made available to all processors. However, making the same data available to all processors may be non-trivial in a distributed environment.

3. Since real systems do not use shared data, protocol models that rely on shared data may not be accurate representation of the real system. Such models may provide inaccurate simulation results and should be avoided.

If your model depends on global state, make sure that the state is consistent on all partitions. If the state does not change after initialization (or is only used during initialization), try to have all partitions generate the same state deterministically. One way to achieve this is to use information contained in the configuration file.

B.1.2 Accessing Other Nodes

A node should not access another node's data structure because the two nodes may be in different partitions. In particular, programmers may be tempted to have one node update another node's statistics or applications. This should be avoided: statistics should be updated by the node to which the statistics belong.

If, for some reason. a node is required to access another node's data structure, then the code should check if the two nodes are in different partitions and take appropriate action if they are.

Consider the code segment in Figure B-1, which could be part of the function that initializes applications. This code segment checks if the originating node (with node identifier `sourceNodeId`) is on the local partition and aborts if it is not.

```
...
node = MAPPING_GetNodePtrFromHash(nodeHash, sourceNodeId)
if (node == NULL)
{
   sprintf(errorString,
        "%s: Node %d does not exist",
        appInput.inputStrings[i],
        sourceNodeId);
    ERROR_ReportError(errorString);
}
...
```

FIGURE B-1. Handling Nodes in a Remote Partition: Incorrect Way

However, it is more appropriate to not abort if the originating node is not in the local partition and to continue processing as long as the node exists in a remote partition. Figure B-2 shows how this is done in function APP_InitializeApplications by using function PARTITION_NodeExists to check whether the node exists at all. If the node exists but is in another partition, then the function continues without taking any action. APP_InitializeApplications is implemented in EXATA_HOME/main/application.cpp.

```
void
APP_InitializeApplications(
    Node *firstNode,
    const NodeInput *nodeInput)
{
  ...
    if (firstNode == NULL)
        return; // this partition has no nodes.

    nodeHash = firstNode->partitionData->nodeIdHash;
    ...
    for (i = 0; i < appInput.numLines; i++)
    {
        sscanf(appInput.inputStrings[i], "%s", appStr);
        ...
        if (strcmp(appStr, "FTP") == 0)
        {
           ...
        }
        else if (strcmp(appStr, "GSM") == 0)
        {
            ...
            // Call Originating(MO) node
            if (PARTITION_NodeExists(firstNode->partitionData, sourceNodeId)
                == FALSE)
            {
               sprintf(errorString,
                    "%s: Node %d does not exist",
                    appInput.inputStrings[i],
                    sourceNodeId);
                ERROR_ReportError(errorString);
            }
            node = MAPPING_GetNodePtrFromHash(nodeHash, sourceNodeId);
            if (node == NULL)
            {
                // not on this partition
                continue;
            }
            ...
       }
       ...
     }
     ...
}
```

FIGURE B-2. Handling Nodes in a Remote Partition: Correct Way

In Figure B-2, the fist parameter of function MAPPING_GetNodeFromHash is set to `firstNode->partitionData->nodeIdHash`. `firstNode->partitionData->nodeIdHash` contains pointers to nodes that are in this partition.

The first parameter of function MAPPING_GetNodeFromHash can alternatively be set to `firstNode->partitionData->remoteNodeIdHash`. `firstNode->partitionData->remoteNodeIdHash` contains pointers to reference nodes which are shadow copies of nodes in other partitions. The reference nodes are used during MAC initialization to ensure that IP addresses are assigned properly. The user should never assume that the reference nodes contain any useful data.

Instead of function MAPPING_GetNodePtrFromHash, function PARTITION_ReturnNodePointer can be used. PARTITION_ReturnNodePointer returns a pointer to the node if the node exists and returns NULL if it does not. To use this function, replace the following line in Figure B-2:

```
node = MAPPING_GetNodePtrFromHash(nodeHash, sourceNodeId);
```

with the following line:

```
PARTITION_ReturnNodePointter(partitionData, &node, sourceNodeId);
```

B.1.3 MAC Lookahead

EXata uses a conservative parallel algorithm. To gain efficiency from parallel execution, protocols estimate a lookahead interval, which is a time interval during which nodes on different processors are guaranteed not to interfere with each other. The longer the lookahead interval, the better the parallel performance.

Some MAC protocols, such as the IEEE 802.11 MAC, set a small lookahead based on the time it takes the physical radio device to change from receiving mode to transmitting mode.

Some MAC protocols, such as the point-to-point link protocol, use the transmission delay as the lookahead.

MAC protocols that are very predictable, such as the TDMA MAC protocol, can make good use of lookahead. TDMA protocols can predict the times of message transmissions exactly because transmissions occur in pre-assigned time slots. This exact time is called the earliest output time (EOT).

A MAC protocol should either set a minimum lookahead for each interface or should indicate that it is not EOT-capable. The minimum lookahead is the smallest delay that is passed to function PHY_StartTransmittingSignal. For optimal performance, the minimum lookahead should be set to the largest possible value that the correct functioning of the protocol allows (which may be 0 in some cases).

The lookahead functions that programmers can use are listed in Table B-1. The prototypes for these functions are contained in EXATA_HOME/include/parallel.h.

TABLE B-1. Lookahead Functions

Function	Description
PARALLEL_SetMinimumLookaheadForInterface	This function sets the minimum delay between messages going out on an interface. This is usually the minimum delay before transmission (ram-up delay) for wireless interfaces, or the transmission delay for wired interfaces.
PARALLEL_SetProtocolIsNotEOTCapable	This function is used to indicate that the protocol is not capable of setting a specific EOT.
PARALLEL_AddLookaheadHandleToLookaheadCalculator	This function adds a new lookahead handle to the lookahead calculator.
PARALLEL_SetLookaheadHandleEOT	This function updates the earliest output time.

Figure B-3 shows how the minimum lookahead is set in the IEEE 802.11 MAC protocol function MacDot11Init, which is implemented in EXATA_HOME/libraries/wireless/src/mac_dot11.cpp.

```
void MacDot11Init(
    Node* node,
    int interfaceIndex,
    const NodeInput* nodeInput,
    PhyModel phyModel,
    SubnetMemberData* subnetList,
    int nodesInSubnet,
    int subnetListIndex,
    NodeAddress subnetAddress,
    int numHostBits,
    BOOL isQosEnabled,
    NetworkType networkType,
    in6_addr *ipv6SubnetAddr,
    unsigned int prefixLength)
{
    BOOL wasFound;
    ...
    MacDot11TraceInit(node, nodeInput, dot11);

#ifdef PARALLEL //Parallel
    PARALLEL_SetProtocolIsNotEOTCapable(node);
    PARALLEL_SetMinimumLookaheadForInterface(node,
                                             dot11->delayUntilSignalAirborn);
#endif //endParallel
    ...
}
```

FIGURE B-3. Setting Minimum Lookahead

If your model's behavior is very predictable, or if the earliest time at which your protocol will send a response changes during the protocol's operation, you must allocate a lookahead handle and then repeatedly set the earliest output time.We show how this is done for the TDMA MAC protocol in Figure B-4 and Figure B-5. The lookahead handle is allocated in the function TDMA initialization function, MacTdmaInit, as shown in Figure B-4. Every time function MacTdmaInitializeTimer is called, the earliest output time is updated, as shown in Figure B-5. Functions MacTdmaInit and MacTdmaInitializeTimer are implemented in EXATA_HOME/libraries/wireless/src/mac_tdma.cpp.

```
void MacTdmaInit(Node* node,
                 int interfaceIndex,
                 int interfaceAddress,
                 const NodeInput* nodeInput,
                 const int subnetListIndex,
                 const int numNodesInSubnet)
{
    BOOL wasFound;
   ...
#ifdef PARALLEL //Parallel
    tdma->lookaheadHandle = PARALLEL_AllocateLookaheadHandle (node);
    PARALLEL_AddLookaheadHandleToLookaheadCalculator(
        node, tdma->lookaheadHandle, 0);
#endif //endParallel

    MacTdmaInitializeTimer(node, tdma);
}
```

FIGURE B-4. Setting Lookahead Handle

```
static
void MacTdmaInitializeTimer(Node* node, MacDataTdma* tdma) {
    int i;
    ...
    if (i == tdma->numSlotsPerFrame) {
       ...
      }
    else {
        Message *timerMsg;
        clocktype delay;
        delay = (tdma->slotDuration + tdma->guardTime) * i
                 + tdma->interFrameTime;
        ...
#ifdef PARALLEL //Parallel
        PARALLEL_SetLookaheadHandleEOT(node,
            tdma->lookaheadHandle,
            getSimTime(node) + delay + 5000);
#endif //endParallel
        ...
    }
}
```

FIGURE B-5. Setting Earliest Output Time

B.1.4 Inter-Layer APIs

Circumventing the inter-layer APIs (see Section 4.11) violates the minimum lookahead settings. For this reason, do not violate protocol layer APIs by sending messages directly to other nodes. Use the protocol stack to send the message through the network.

B.2 External Interface Issues

This section lists some issues which often arise when developing parallel-safe code for use with external interfaces.

B.2.1 Node Lists

If a program uses code that iterates among all nodes (e.g., by using the field `firstNode` of `partitionData`), then in a sequential run the code will iterate through all nodes in sequence (assuming default partitioning). In parallel execution, each partition has a subset of the complete list of nodes. So, if the code acquires nodes by examining or iterating through the list of nodes, the results from a sequential run may be very different from results from a parallel run.

B.2.2 Loose Events

Certain events to be scheduled for a node (which may be on a different partition) can be scheduled with a best-effort time delay.

We illustrate this by taking as an example the implementation of the Forward application. A Forward application instance at both the source node and the destination node needs to be created before the real TCP packets arrive at the destination. To accomplish this, the Forward application at the source node sends an application instantiation message to the destination node before sending any real TCP packets. This message is not real network traffic and serves to trigger the destination node to create a Forward application instance. This instantiation message is sent with a 0 delay value and a loose, or best-effort, scheduling requirement. The Forward application is implemented in files app_forward.h and app_forward.cpp in EXATA_HOME/libraries/developer/src.

Figure B-6 shows an example of how the instantiation message is sent in function AppLayerForwardBeginExternalDataTCP, which is implemented in file app_forward.cpp.

```
void AppLayerForwardBeginExternalDataTCP(
    NodeAddress from,
    Node * nodeA,
    NodeAddress nodeIdA,
    NodeAddress addressA,

    NodeAddress to,
    Node * nodeB,
    NodeAddress nodeIdB,
    NodeAddress addressB,

    char *data,
    int dataSize,
    int interfaceId,
    char * interfaceName
    )
{
    AppDataForward *forward;
    ...
    forward = AppLayerGetForward(
        nodeA,
        addressA,
        addressB,
        interfaceId);
    if (forward == NULL)
    {
        Message* instantiateMessage;
        EXTERNAL_ForwardInstantiate *instantiate;
        ...
        instantiateMessage = MESSAGE_Alloc(
            nodeB,
            APP_LAYER,
            APP_FORWARD,
            MSG_EXTERNAL_ForwardInstantiate);
        ...
        // Send the message with no delay
        EXTERNAL_MESSAGE_SendAnyNode (nodeB->partitionData, nodeB,
            instantiateMessage, 0, EXTERNAL_SCHEDULE_LOOSELY);
    }
    ...
}
```

FIGURE B-6. Scheduling Loose Events

B.2.3 Partition Communication

Partition communication allows one partition to send a message to a different partition. To use partition communication, the external interface code registers a communication handle as part of its initialization work. A name and a function that will be called for processing communication messages are provided when the registration function is called by the external interface. In contrast to a normal message, these partition communication messages can then be sent to the function that was registered.

The partition communication functions that programmers can use are listed in Table B-2. The prototypes for these functions are contained in EXATA_HOME/include/partition.h.

TABLE B-2. Partition Communication Functions

Function	Description
PARTITION_COMMUNICATION_RegisterCommunicator	This function allocates a message identifier and registers the handler that is invoked to receive callbacks when messages with that identifier are sent.
PARTITION_COMMUNICATION_FindCommunicator	This function locates an already registered communicator.
PARTITION_COMMUNICATION_SendToPartition	This function transmits a message to a specific partition.
PARTITION_COMMUNICATION_SendToAllPartitions	This function transmits a message to all partitions.

B.2.4 Forwarding Packets to External Interfaces

Function EXTERNAL_RemoteForwardData is similar to the function EXTERNAL_ForwardData, except that it forwards the message to an external interface that is on a remote partition. Each partition has its own set of interfaces and a message can be forwarded back to an external interface on a different partition. In this case, the message to be forwarded is copied to the specified partition and then the interface's forward function is invoked on that partition. The message to be forwarded must be *flat*, i.e., it can not contain pointers because those areas of memory are only present on the original partition.

Functions EXTERNAL_ForwardData and EXTERNAL_RemoteForwardData are implemented in files EXATA_HOME/include/external.h and EXATA_HOME/main/external.cpp.