COMPUTER NETWORK SIMULATION USING OMNET++
(A A Practical Approach)

By:
Muhammad Azhar Iqbal
Muhammad Arshad Islam
Muhammad Aleem
Hassan Murtaza

HIGHER EDUCATION COMMISSION
ISLAMABAD – PAKISTAN
All rights are reserved. No part of this publication may be reproduced, or transmitted, in any form or by any means – including, but not limited to, electronic, mechanical, photocopying, recording, or, otherwise or used for any commercial purpose what so ever without the prior written permission of the publisher and, if publisher considers necessary, formal license agreement with publisher may be executed.

Project: “Monograph and Textbook Writing Scheme” aims to develop a culture of writing and to develop authorship cadre among teaching and researcher community of higher education institutions in the country. For information please visit: www.hec.gov.pk

HEC – Cataloging in Publication (CIP Data):

Iqbal, Muhammad Azhar.

Computer Network Simulation using OMNeT++ (A Practical Approach). / Dr. Muhammad Azhar Iqbal… (et. al.).


I. TITLE. II. Muhammad Arshad Islam. III. Muhammad Aleem. IV. Hassan Murtaza.

005.4476--ddc23.

First Edition: 2017
Copies Printed: 500
Published By: Higher Education Commission – Pakistan
Disclaimer: The publisher has used its best efforts for this publication through a rigorous system of evaluation and quality standards, but does not assume, and hereby disclaims, any liability to any person for any loss or damage caused by the errors or omissions in this publication, whether such errors or emissions result from negligence, accident, or any other cause.
# TABLE OF CONTENTS

PREFACE ........................................................................................................................................ VII

ACKNOWLEDGEMENT ......................................................................................................... IX

CHAPTER 1 .................................................................................................................................. 1

**Computer Network Fundamentals** .................................................................................... 1
1.1 Introduction ...................................................................................................................... 1
1.2 Building Blocks of Computer Networks ......................................................................... 2
1.3 Taxonomy of Computer Networks .................................................................................. 4
   1.3.1 Network Connectivity ............................................................................................... 4
   1.3.2 Application Paradigm ............................................................................................... 4
   1.3.3 Network Medium ..................................................................................................... 5
   1.3.4 Network Coverage .................................................................................................. 6
   1.3.5 Ownership ............................................................................................................. 6
1.4 Point-to-Point Network Issues and Simulation ............................................................... 6

Review Questions ..................................................................................................................... 7

CHAPTER 2 .................................................................................................................................. 9

**Scientific Methods and Simulation Tools** ......................................................................... 9
2.1 Introduction ........................................................................................................................ 9
2.2 Simulation Models ............................................................................................................ 10
   2.2.1 Time Presence ......................................................................................................... 11
   2.2.2 Value type ............................................................................................................... 11
   2.2.3 Simulation behavior ............................................................................................... 11
2.3 Discrete-Event Simulation ............................................................................................. 11
   2.3.1 Trace-driven simulation ......................................................................................... 11
   2.3.2 Time-driven simulation ......................................................................................... 11
   2.3.3 Event-driven simulation ......................................................................................... 12
2.4 Organization of DES ...................................................................................................... 12
   2.4.1 System state ............................................................................................................ 12
   2.4.2 Simulation clock ..................................................................................................... 12
   2.4.3 Event list ................................................................................................................ 12
   2.4.4 Statistical counters ................................................................................................. 12
   2.4.5 Initialization routine ............................................................................................... 12
   2.4.6 Timing routine ........................................................................................................ 12
   2.4.7 Event routine ......................................................................................................... 12
   2.4.8 Library routines ...................................................................................................... 12
   2.4.9 Report generator ................................................................................................... 12
   2.4.10 Main program ....................................................................................................... 12
2.5 Steps in a Simulation Study ............................................................................................. 12
2.6 Comparison of Network Simulators .............................................................................. 13
2.7 Advantages of Simulation .............................................................................................. 14
2.8 Disadvantages of Simulation ......................................................................................... 14

Review Questions ..................................................................................................................... 15

CHAPTER 3 .................................................................................................................................. 17

**Introduction to OMNET++** ................................................................................................. 17
3.1 Introduction ...................................................................................................................... 17
3.2 Simulation Modeling in OMNET++ ................................................................................ 19
3.3 OMNET++ Network Simulation Structure ..................................................................... 19
3.3.1 NED Language and Structure of .ned files ............................................. 20
3.3.2 Algorithm Implementation and Structure of the .cc files .......................... 21
3.3.3 Configuration Options and Structure of the .ini files ................................. 22

CHAPTER 4 .................................................................................................................. 25
GENERAL STRUCTURE OF OMNET++ PROJECT .................................................. 25
4.1 THE OMNET++ PROJECT ................................................................................. 26
4.2 BUILDING AN OMNET++ PROJECT ................................................................. 26
PRACTICE TASKS ...................................................................................................... 36

CHAPTER 5 .................................................................................................................. 39
IMPLEMENTATION OF LAYERED ARCHITECTURE ............................................. 39
5.1 OVERVIEW ......................................................................................................... 40
5.2 TCP/IP ARCHITECTURE ..................................................................................... 40
5.3 LAYERED ARCHITECTURE IN OMNET++ ..................................................... 40
5.4 BUILDING A LAYERED ARCHITECTURE IN OMNET++ ............................... 40
PRACTICE TASKS ...................................................................................................... 59

CHAPTER 6 .................................................................................................................. 61
IMPLEMENTATION OF CIRCULAR REDUNDANCY CHECK (CRC) ...................... 61
6.1 ERROR DETECTION ............................................................................................ 62
6.2 CYCICAL REDUNDANCY CHECK (CRC) ......................................................... 62
6.3 IMPLEMENTATION OF CRC IN OMNET++ .................................................... 62
PRACTICE TASK ....................................................................................................... 71

CHAPTER 7 .................................................................................................................. 73
IMPLEMENTATION OF ARQ STOP-AND-WAIT ................................................... 73
7.1 RELIABLE TRANSMISSION – AN OVERVIEW .............................................. 74
7.2 STOP-AND-WAIT APPROACH ........................................................................ 74
7.3 IMPLEMENTATION OF ARQ STOP-AND-WAIT .......................................... 75
PRACTICE TASK ....................................................................................................... 84

CHAPTER 8 .................................................................................................................. 85
IMPLEMENTATION OF SLIDING WINDOW ............................................................ 85
8.1 SLIDING WINDOW ALGORITHM ...................................................................... 86
8.2 WORKING OF SLIDING WINDOW ALGORITHM ............................................ 86
8.3 IMPLEMENTATION OF SLIDING WINDOW ALGORITHM ............................... 88
PRACTICE TASK ....................................................................................................... 97

CHAPTER 9 .................................................................................................................. 99
IMPLEMENTATION OF MULTIPLEXING/DEMULTIPLEXING ............................... 99
9.1 MULTIPLEXING AND DEMULTIPLEXING IN TCP/IP STACK .......................... 100
9.2 IMPLEMENTATION OF MULTIPLEXING/DEMULTIPLEXING .......................... 100
PRACTICE TASK ..................................................................................................... 115

CHAPTER 10 ............................................................................................................... 117
GUI INTERFACES AND ENHANCEMENTS ......................................................... 117
10.1OMNET++ EXECUTION ENVIRONMENT ......................................................... 118
10.2COMMAND-LINE INTERFACE (CMENV) ...................................................... 118
10.3TkENV (TCL/TK-BASED) INTERFACE ............................................................. 119
10.4QtENV INTERFACE ......................................................................................... 119
10.5EDITING OPTIONS OF NED IN GRAPHICAL MODE ...................................... 120
10.6GRAPHICAL/Visualization ENHANCEMENTS .......................................... 120
10.6.1 Use of Display Strings .............................................................................. 120
10.6.2 Use of setTagArg() - Change in Display Strings at Runtime .................... 128
Objectives

In our experience, network modeling and simulations are complex tasks for research students. Moreover, modeling tools such as network simulators are difficult to learn and practice and require advanced level expertise. Network simulation tools are helpful in two ways: first, provisioning of a virtual environment that mimics a real-life scenario and secondly, it provides integrated development environment for the evaluation of proposed algorithms before actual implementation on real-world devices. A number of network simulation tools (i.e., NS-2, NS-3, OMNeT++, OPNET, QualNet, GloMoSim, JiST/SWANS, DIVERT, NCTUns, etc.) [1] [2] based on sequential/parallel Discrete Event Simulation (DES) kernel are available. However, the selection of network simulator depends on several factors for example ease of use, the learning curve of the programming language involved, type of scenario one may intend to simulate etc. Keeping these factors in view, we have identified that OMNeT++ is among the most popular DES simulators attracting fairly a large researcher community [3].

OMNeT++ is available in two editions i.e., Academic and Commercial [4]. OMNeT++ Academic Edition is an open, extensible, modular, component-based simulation library and framework, primarily used for modeling and performance analysis of network (including wired and wireless) technologies. The commercial version of the OMNeT++ (that is known as OMNEST [5]) has broader capabilities designed for the enterprise IT environment and additionally provides documentation and professional support. Considering the increasing demands of OMNeT++, we have formulated this practical simulation guide for the beginners who are interested in learning computer network simulations.

The objective of this book is to provide a good starting point for students that have basic prior knowledge of network simulations. This book is an effort to partially fill the gap associated with the understanding of computer network concepts through programming in an abstract way. We believe that after performing proposed network simulation experiments, students will find it easier to understand and use available frameworks (of OMNeT++) with implemented concepts of the computer network at different layers of TCP/IP protocol stack.

Key Feature

The uniqueness of this book is that it covers simulation details along with the theoretical concepts of computer networks. The key feature of this book is that it targets various aspects of TCP/IP stack to provide its readership a better perspective both in terms of simulation skills as well as the basic understanding of TCP/IP. This book considers the implementation of layering perspective of TCP/IP that (to the best of our knowledge) has not been addressed in the available OMNeT++ text.

Audience

This book is suitable for graduate students enrolled in computer network course. This book assumes that the reader has a good understanding of OMNeT++ TicToc tutorial for simple networks (without compound module implementation). Instructors are facilitated in this book in a comprehensive way to explain computer network theory besides guidance of developing skills in OMNeT++ simulation tool.
Approach

Concerning few simple concepts of TCP/IP stack, this book can be considered as the first attempt to learn the use of OMNeT++ while starting compound modules. This book contains few network simulation experiments describing fundamental simulation steps at an abstract level to implement TCP/IP concepts in OMNeT++. The experiments in this book require intermediate programming skills in C++ as a prerequisite. The codes used in various examples of this book are available online at www.pcn.net.pk.

Organization of the Book

To address the issues of computer network and simulation together, this book is structured as follows:

The first two chapters of this book provide an abstract-level overview of computer network fundamentals, simulations methodologies, and corresponding issues. This is followed by the concise introduction to OMNeT++ simulator in Chapter 3. These chapters build foundation blocks of the computer network simulation and are considered pre-requisites for the following chapters.

The primary focus of Chapter 4 is to establish the understanding of the general structure of OMNeT++ project along with the sequence of necessary steps for building an OMNeT++ project. In addition, configuration requirements and implementation details of topology description files are discussed.

The contents of Chapter 5 are oriented along the lines of TCP/IP layering model with the focus on aspects related to the design and implementation of compound modules in OMNeT++. Simulation steps of this chapter explain the data flow pattern at each layer of TCP/IP stack through the use of network description files and respective implementation details in C++.

Chapter 6 describes the basic concepts of error detection using CRC implementation. In continuation, both Chapter 7 and Chapter 8 explain theoretical as well as the simulation details of the Stop-and-Wait mechanism of Automatic Repeat Request (ARQ) and Sliding Window Algorithm for reliable transmission and flow control, respectively.

Chapter 9 discusses the concept and implementation of multiplexing and demultiplexing in TCP/IP stack.

Chapter 10 presents sophisticated features of OMNeT++ related to the GUI and visualization in network description and corresponding implementation files.

In Chapter 11, we present the details of the case study related to the working of client-server architecture implemented in OMNeT++ (known as DYNA).
We want to appreciate the efforts of the reviewing team at Higher Education Commission (HEC), Pakistan for providing us the feedback and opportunity to publish this book. We would like to acknowledge the cooperation extended by Prof. Dr. Mansoor Ahmed (Vice Chancellor, Capital University of Science and Technology (CUST), Islamabad), Prof. Dr. Muhammad Abdul Qadir (Dean Faculty of Computing, CUST), and Prof. Dr. Nayyer Masood (HoD Computer Science, CUST). We would also like to thank our colleagues at CUST for their continuous moral support and guidance. Finally, we want to acknowledge the most important contribution by our families for showing patience and understanding for the time we spent away from them while writing this book.
Computer Network Fundamentals

Learning Objectives:

After studying this chapter, you will be able to describe:

- the basics of computer networks
- the building blocks of a computer network
- the taxonomy of computer networks
- the standard network architectures
- the functionality at each layer of OSI and TCP/IP network architectures
1.1 Introduction

Two or more autonomous computers connected through guided (network cables) or unguided (air) medium constitute a typical computer network. Abstraction of computer network’s programmable hardware and software generality distinguishes it from other types of networks [6]. The fundamental objectives of computer networks include the sharing of information/resources and to facilitate communication in a cost-effective way. Therefore, we can find a computer network in any modern-day organization or business for the purpose of record keeping of employees and production, inventories and do the payroll. At first instance, these processes may appear separate, however, organizations need to connect them to be able to extract and correlate information to obtain a complete picture. Moreover, computer networks enable organizations to efficiently utilize resources by sharing them. An obvious and widespread example is having a group of office workers share a common printer. Another important aspect of an organization could be the distributed location of their services. A product manager in city Islamabad may feel the need to communicate and share data with the sales person in Karachi on regular basis. In this scenario, computer networks play a vital role by abstracting the remote nature of resources and provide a seamless access to resources similar to the use of the locally available resources. Figure 1.1 represents an example of a typical network within an organization comprising of webservers, file servers, hardware resources etc.

![Figure 1.1: A typical computer network](image)

1.2 Building Blocks of Computer Networks

Basic building blocks of computer networks consist of Hardware and Network Software Architecture [6] [7] (as shown in Figure 1.2).

Concerning hardware building blocks, the network edge refers to all end systems (i.e., hosts, clients, servers, peers etc.) that utilize the network. However, the network core consists of intermediate nodes (i.e., switches, routers, gateways etc.), which are actually required to implement the network. Switches, routers, and gateways are devices or technological boxes to which machines, printers, and other devices are connected. Nowadays, switches/routers are the more common technology to build small/large-scale computer networks. Links either guided (i.e., twisted pair, coaxial cable, fiber optic etc.) or unguided (i.e., microwave, radio waves, infrared etc.) are used to connect edges and core together [7]. The suitability of links within a network depends upon various factors such as link reachability, static or moving nature of an edge, cost of deploying etc.

Software components of network architecture are used to define the standards and techniques for designing and building of communication systems for computers and other intermediate devices [8]. These standards reduce the design complexity of a network architecture.
using layers or levels that are stacked on each other. The basic motivation in designing these layers is to divide the functionality of the network system into small modules. Each layer adds to the functionality offered by the layers beneath. Users mostly interact with the topmost layer of the architecture to access all the functionalities of the lower layers.

Two important network architecture models widely discussed in the literature are Open System Interconnection (OSI) [9] and Transmission Control Protocol/Internet Protocol (TCP/IP) suite [10]. The OSI is a reference model and consists of the seven layers. On the other hand, the
TCP/IP is an example of an implemented model consisting of five layers. Figure 1.3 provides detail of each corresponding layer of the OSI and the TCP/IP suite. Layering provides modularity and the decomposition of the complex problem of building computer networks. Fundamental differences between the OSI and the TCP/IP models are shown in Table 1.1 [11].

<table>
<thead>
<tr>
<th>OSI</th>
<th>TCP/IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seven-Layer model</td>
<td>Five-Layer model</td>
</tr>
<tr>
<td>Reference or Conceptual model</td>
<td>Internet implemented model</td>
</tr>
<tr>
<td>Strict boundaries for protocols</td>
<td>Protocol boundaries not strictly defined</td>
</tr>
<tr>
<td>Standardization before Implementation</td>
<td>Implemented and then adopted as standard</td>
</tr>
</tbody>
</table>

1.3 **Taxonomy of Computer Networks**

Computer networks can be categorized in various ways. Below are the few important aspects that are used to categorize computer networks in literature:

- Network Connectivity
- Application Paradigm
- Network Medium (Link Nature)
- Network Coverage Area
- Ownership

1.3.1 **Network Connectivity**

On the basis of connectivity, computer networks can be categorized into two broad classes 1) *directly connected* and 2) *indirectly connected* computer networks [6] (as shown in Figure 1.4). The directly-connected computer networks are of two types 1) *Point-to-Point* and 2) *Multiple Access*. In the Point-to-Point network, physical medium is limited to a pair of hosts and in Multiple Access networks, several hosts share a common link. On the other hand, indirectly connected computer networks (also called *switched networks*) with one or more intermediate nodes (such as routers, switches, bridges, gateways) can be classified as *Circuit-switched* and *Packet-switched* networks. Circuit-switched networks are used to transfer bit-stream on a static, pre-established network route. On the other hand, in the packet-switched networks, the route is not determined and individual data packet dynamically finds a path towards the destination host. The circuit-switched networks are employed by traditional telephonic systems and the packet-switched networks are an example of real computer networks. A packet-switched network can be classified into *virtual-circuit* based or *datagram* based networks.

1.3.2 **Application Paradigm**

Considering application paradigms [7], we can divide all computer networks into two categories 1) *Peer-to-Peer* [12] [13] and 2) *Client/Server-based* [14] networks. Peer-to-peer architecture (also known as a workgroup) connects a group of computers to enable users to share resources and information. This type of architecture lacks a central authority responsible for storing data and controlling access to different network resources. In peer-to-peer networks, it is the responsibility of the individual users to maintain information about the location of network resources. This characteristic makes peer-to-peer architecture difficult to manage for any organization as issues related to data inconsistency arise due to multiple versions of the same data.
1.3.3 Network Medium

A network that uses any wired medium of communication e.g., Ethernet cable, coaxial cable etc. is considered a wired network. Wired networks are difficult to deploy; however, they offer more control over the communication. Enterprises usually deploy wired networks, as it requires a physical connection to the network. A wireless network utilizes antenna technology and air (such as radio waves, microwaves, infrared etc.) as a medium of communication. Wireless networks are relatively easy and economical to deploy and maintain; however, a strong authentication and encryption frameworks are required to secure the access to the network and maintain data security. Wireless networks can be categorized on the basis of topology, mobility, and placement as shown in Figure 1.5.

In wireless networks, when we consider the topology, we can categorize computer networks into two types, 1) infrastructure-based and 2) infrastructure-less (Ad hoc) [15]. A controlling authority, usually known as Access Point (AP) or Base Station (BS) is required for operations of an infrastructure-based network. The AP offers several important advantages over
an ad-hoc network e.g. enhanced level of security, faster transmission speed, and integration with other existing networks. Contrary, each device in an ad-hoc network can communicate directly with each other. The central authority AP or BS that controls the communication is absent in this case, which results in a less sophisticated security as compared to an infrastructure-based network.

In terms of mobility, fixed and mobile networks are two fundamental types of wireless networks. Fixed wireless networks comprise of devices that are installed at fixed location such as homes and offices. Fixed wireless devices usually need a permanent electrical power source. Mobile wireless or portable wireless devices, however, can operate using batteries that are charged periodically. The bandwidth of the mobile networks is not as good as of the fixed wireless devices and such systems are usually used, as a backup in case fixed systems are not operational.

With respect to the placement, wireless networks can be categorized as Terrestrial and Satellites based. In the terrestrial networks, all communication is done within the earth atmosphere. In the satellite networks, signals are beamed up to orbiting satellites in the case of satellite communications that forward received signals to the destination on the ground. This model is used when deploying a terrestrial network is not easy and areas on earth cannot be covered by terrestrial wireless due to economic reasons. The delay incurred in the case of the terrestrial network is far less as compared to the satellite communication.

1.3.4 Network Coverage

While considering the area of coverage a wired network can be Local Area Network (LAN), Metropolitan Area Network (MAN), and Wide Area Network (WAN). Similarly, a wireless network can be a Wireless Body Area Network (WBAN), Wireless Personal Area Network (WPAN), Wireless Local Area Network (WLAN), Wireless Metropolitan Area Network (WMAN) and Wireless Wide Area Network (WWAN). Short description with a range of such networks is given below:

- LAN – Network that spans in a small area (approx. 1000m)
- MAN – Network that spans in a relatively large area (approx. 50 km)
- WAN – Network that spans a large area (approx. 1000Km)
- WBAN – Network of biomedical sensor nodes on human body (approx. few cm)
- WPAN – Network of short range devices i.e., Zigbee, Bluetooth (approx. 10 m)
- WLAN – Network allows limited user connectivity in limited area (approx. 100m)
- WMAN – Multiple networks’ connected network in a metropolitan (approx. 1 km)
- WWAN – Network maintained over large area (approx. 10 km)

1.3.5 Ownership

Computer networks can be built as public or private. As the name suggests, a public network is accessible to the general public and anyone can connect to the network. Examples of a public network include network available at coffee shops, airports, and other public places. A private network is owned/built and operated by an organization or individual who may decide to set restriction and rules to regulate the access to the network. Both wired and wireless networks can be built as a public or private network.

1.4 Point-to-Point Network Issues and Simulation

The simplest computer network is a Point-to-Point network, where two end-nodes (computers, servers, hosts, peers etc.) are connected via direct link (either through wired or wireless medium). The reason behind the simplicity of Point-to-Point network lies in the absence of intermediate components that may introduce complexity in terms of connectivity and working of nodes. In general, the issues involved in Point-to-Point network exist in all types of directly and indirectly connected networks.

In our opinion, Point-to-Point network simulation is a good first step to learn basic concepts (such as packet transmission/reception, packet flow control, error detection, multiplexing and
demultiplexing) at different layers of TCP/IP architecture. Therefore, this book presents simulation experiments focusing a Point-to-Point network.

Review Questions
1. Explain what do you understand by a computer network? and How would you differentiate a computer network from other types of available networks?
2. Narrate the most fundamental objectives of computer networks.
3. List down the network hardware devices that serve as building blocks of a computer network. How can network hardware devices be categorized?
4. What kind of software architecture is preferred to integrate hardware building blocks into an effective communication service?
5. How would you differentiate between the OSI and the TCP/IP architecture?
6. Explain the taxonomy of wireless computer networks?
7. What is a Point-to-Point computer network? Why a Point-to-Point network considered as good starting point to learn basic concepts of computer networks?
8. How would you differentiate between Point-to-Point and packet-switched computer networks?
Scientific Methods and Simulation Tools

Learning Objectives:

After studying this chapter, you will be able to describe:

- the classification of simulation models
- the basics of Discrete Event Simulation (DES)
- the variants of DES
- the typical components of DES
- the difference between Terminating and Steady-State Simulations
- the taxonomy of network simulation tools
- the steps in network simulation study
- the advantages and disadvantages of network simulation
2.1 Introduction

Computer simulation represents a computer-based environment that resembles a realistic model of computer and communication infrastructure. Computer simulation can be employed in all major real-life fields such as business, physics, chemistry, biosciences, communications, distributed computing, computer networks, etc. [16]. There are several ways to study a real-life system [17] as shown in Figure 2.1.

![Figure 2.1: Ways to study a real-world system](image)

Like any other system, computer network systems can also be studied in different ways as shown in Figure 2.1. However, the successful deployment of a new communication system or computer network always relies on the predicted behavior of a system’s performance already obtained through the use of experimental, analytical, or simulation techniques. Experimental techniques are expensive and analytical methods are unable to fully grasp the characteristics associated with communication and computing networks. Therefore, currently, simulation techniques are used extensively to reduce the utilization of expensive experimentation during the design and performance analysis of all types of computer networks [18].

2.2 Simulation Models

A simulation model is a realistic representation of a system that is required to understand the working and performance evaluation of a system [19] [20].

![Figure 2.2: Simulation models' properties](image)
A number of simulation models have been proposed in the literature. Generally, a simulation model can be classified based on three properties [17]: time presence, value-type, and simulation behavior as shown in Figure 2.2.

2.2.1 Time Presence
Time presence shows whether a simulation model incorporates time-factor in the simulation process or not. Accounting time factor in the simulation process is known as a dynamic simulation in contrast to a static simulation where time is not considered as a contributing component.

2.2.2 Value type
A simulation model can be based on one of the two value types: discrete or continuous. A discrete simulation model only employs values from a certain finite range. In the discrete simulation model, changes in values occur spontaneously with respect to points in time. In contrast to the discrete simulation model, a continuous model may employ values from an infinite range. In continuous model, changes in variables occur regularly with respect to time and can be modeled as differential equations.

2.2.3 Simulation behavior
Simulation behavior shows the process of simulation. A deterministic model does not have a random part (neither the random events nor the random values are generated). This means that if we repeat the simulation process by employing the same parameters every time the results (values produced and sequence of events) will be same. A probabilistic model has a random component resulting in possibly different results for the same repeated simulation.

Based on the above-mentioned simulation properties, we can classify every simulation. For example, a simulation to generate a network-path between a host and destination machine represents a simulation process based on dynamic, discrete, and probabilistic properties [17]. However, simulating a rocket's trajectory by employing initial velocity with fixed wind resistance represents a dynamic, continuous and deterministic simulation. Based on the three model types (shown in Figure 2.2), the combination of the simulation properties results in several kinds of simulations. The dynamic, discrete, and probabilistic model are often used to simulate the process of complex systems such as computer networks. This model is often referred as Discrete-Event Simulation that comprises of well-defined ordered events (a specific change in system state with respect to time).

2.3 Discrete-Event Simulation
Discrete-Event Simulation (DES) [16] represents a mathematical and logical model of a real physical system that evolves over time. DES simulation model is highly suited to the real complex systems such as computer networks where a state change occurs at explicit time intervals. There are several variants of DES such as trace-driven simulation, time-driven, and event-driven simulations.

2.3.1 Trace-driven simulation
In trace-driven simulation, real systems traces are collected and employed instead of synthetic input sources (e.g., random number generators). A trace-driven DES represents a system consisting of events based on real traces. A user can modify a certain system trace to observe the resulting system behavior and control the simulation process.

2.3.2 Time-driven simulation
A time-driven DES represents a simulation process where a fixed time interval is used to activate events. This type of DES simulation is useful for modeling a time-dependent event modeling.
2.3.3 **Event-driven simulation**
Event-driven DES is useful to model systems that produce events at irregular intervals. An event-driven simulation process is more time efficient compared to the time-driven model that always steps through the intervals where even no event takes place.

2.4 **Organization of DES**
The typical components a DES model comprises of are listed and described below:

2.4.1 **System state**
*System state* comprises a collection of variables used to represent collectively the system’s outlook at certain time instance.

2.4.2 **Simulation clock**
*A simulation clock* is a variable that contains the elapsed time for the current simulation process.

2.4.3 **Event list**
An *event list* is a time-based event queue containing the next time when each type of the event will take place.

2.4.4 **Statistical counters**
System performance and accounting information are stored in variables known as *statistical counters*.

2.4.5 **Initialization routine**
*Initialization routine* represents a function or subprogram that initializes the simulation environment at time-unit 0.

2.4.6 **Timing routine**
*A timing routine* is a time scheduling process that increments the simulation-clock so that certain events could take place (waiting for that time instance).

2.4.7 **Event routine**
For each event-type, there is a separate event routine. An *event-routine* is responsible for updating the system-state after the occurrence of the event.

2.4.8 **Library routines**
DES comprises of several *library routines* (set of subprograms) used to generate random simulation observations from the probability distribution based on the employed simulation model.

2.4.9 **Report generator**
*The report generator* is a sub-program that utilizes the usage accounting information, estimates and measures several performance-related attributes to generate reports at the end of the simulation.

2.4.10 **Main program**
The main coordinating sub-program of the simulation is referred as the *main program*. It invokes the timing routines, updates the system state, and invokes the report-generator at the end of the simulation.

2.5 **Steps in a Simulation Study**
A simulation-based system study is a multi-step activity where each step represents a certain task that has to be performed to achieve the overall simulation goal [21]. Figure 2.3 shows some of the desired steps in a simulation study.
Step 1 shown in Figure 2.3 represents the formulation of underlying simulation problem related to a real system. This step is required so that the problem to be simulated should first be understood clearly. In the next step (shown as Identifying objectives and plans), the objectives of the study have to be finalized. Objectives represent all the questions to be answered by the simulation study. Moreover, in this step, it is also determined whether the simulation is the best available method or not for the study. Step 3 represents the abstractions related to the simulation model for the real system. The abstractions are useful for the study of the complex real system and help to make assumptions for the unknown attributes of the modeled system. Data collection (step 4) represents an important activity, which can consume large time depending on systems complexity. Step 5 represents translating a conceptual simulation model into the computer-based program. The effort needed to convert the conceptual model into the computer acceptable format depends on the underlying simulation model. After building the simulation model, verification can be carried out to make sure the requirements of the simulations are correctly translated or converted. Moreover, the validation is required to ensure that all the required inputs and assumptions are acceptable to the built simulation model. The process of verification and validation is repeated to make sure that the simulation models accurately mimic the real system. Step 6 consists of designing the experiments. Experiment design involves identification of the significant input parameters (having a greater impact on simulation) and the identification of desired output factors. After designing the experiments, large-scale simulation executions could be carried out to meet the certain desired simulation goals (as shown in step 7). In step 8, all the execution runs could be documented along detailed results for further analysis.

2.6 Comparison of Network Simulators

Due to the unique property of state changing at discrete points in time, DES simulator gain importance in the field of computer networks [16]. A plethora of DES-based computer network simulators have been reported in the literature and it is not possible to include all here. However, based on sequential/parallel DES kernel, renowned network simulators are NS-2 [23], NS-3 [24], OMNeT++ [3], OPNET[25], QualNet [26], Netsim [27], NCTUs [28], etc. The selection of a suitable
network simulator for evaluating research work is an important task for researchers. The suitability of network simulator depends on various contributing factors summarized in Table 2.1.

Table 2.1: Summary of network simulators

<table>
<thead>
<tr>
<th>Factors</th>
<th>NS-2</th>
<th>NS-3</th>
<th>OMNeT++</th>
<th>OPNET</th>
<th>QualNet</th>
<th>NetSim</th>
<th>NCTUns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Nature</td>
<td>Wired</td>
<td>Wired</td>
<td>Wired</td>
<td>Wired</td>
<td>Wired</td>
<td>Wired</td>
<td>Wired</td>
</tr>
<tr>
<td>Simulation Event Type</td>
<td>Discrete-event</td>
<td>Discrete-event</td>
<td>Discrete-event</td>
<td>Discrete-event</td>
<td>Discrete-event</td>
<td>Stochastic Discrete-event</td>
<td>Kernel Reentering method</td>
</tr>
<tr>
<td>High Scalability Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GUI Support</td>
<td>Good</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>Difficult</td>
<td>Difficult</td>
<td>Easy</td>
<td>Easy</td>
<td>Moderate</td>
<td>Easy</td>
<td>Difficult</td>
</tr>
<tr>
<td>Programming Languages</td>
<td>C++ OTCL</td>
<td>C++ Python</td>
<td>C++ NED</td>
<td>C and C++</td>
<td>C++</td>
<td>C and Java</td>
<td>C++</td>
</tr>
<tr>
<td>Parallelism Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Installation and Configuration Time</td>
<td>High</td>
<td>High</td>
<td>Less</td>
<td>Less</td>
<td>Moderate</td>
<td>Less</td>
<td>Less</td>
</tr>
<tr>
<td>License Type</td>
<td>Academic</td>
<td>Academic</td>
<td>Academic Commercial</td>
<td>Commercial</td>
<td>Commercial</td>
<td>Proprietary</td>
<td>Academic Commercial</td>
</tr>
<tr>
<td>Supported Operating Systems</td>
<td>Linux Windows MAC OS</td>
<td>Linux Window</td>
<td>Linux Windows MAC OS</td>
<td>Windows</td>
<td>Linux Unix Windows MAC OS</td>
<td>Windows</td>
<td>Linux</td>
</tr>
</tbody>
</table>

The synthesis of available studies [2] [29] [30] [31] supports the arguments of extensive OMNeT++ usage in researcher community.

2.7 Advantages of Simulation

Some of the major advantages of the simulations are [16] [22]:

- Sometimes simulation is the only way to investigate a real-life system
- Useful to investigate a system’s behavior under constrained set of factors
- Simulations help to compare alternative system designs
- Controlled system behavior could be simulated
- Systems with long time frames could easily be simulated in a short time
- Detailed study can be done using time-based expanded simulations

2.8 Disadvantages of Simulation

Some of the disadvantages of the simulations are [16] [22]:

- Probabilistic simulation model produces only an estimation of the simulated system
- Simulations often represent a complex system with higher difficulty and time consuming to exercise
- Realistic execution of the simulated model often enjoys great user confidence in the results ignoring to their justification
To simulate a realistic complex system, a large amount of memory and processing power is required.

**Review Questions**
1. How would you argue in favor of computer simulations to understand the working of computer networks before actual deployment?
2. Explain the significance of a simulation model? How would you classify simulation models?
3. Why is DES model suitable for network simulation? Briefly, describe the DES variants.
4. Identify and describe the DES components.
5. What are the typical steps involved in a typical network simulation study?
CHAPTER – 3

Introduction to OMNeT++

Learning Objectives:

After studying this chapter, you will be able to describe:

- the strengths of OMNeT++
- simulation modeling in OMNeT++
- typical components of OMNeT++ simulation
- the NED language and structure of NED files
- the object-oriented model of implementation (.CC) files
- the structure and importance of configuration (.INI) files
3.1 Introduction

The Objective Modular Network Testbed in C++ (OMNeT++) is highly modular, extensible, scalable, well-structured, component-based DES, which primarily built on C++ library to simulate computer networks [4]. The fundamental motivation behind OMNeT++ development is related to the provisioning of an influential open-source DES for academic and research-oriented commercial institutions. The secondary objective is to fill the gap between open source (with limited capabilities) and expensive commercial simulators (having classical GUI and rich library of simulation packages) [3] [32]. Primarily, OMNeT++ is not a network simulator itself but the fundamental reason for its far-flung pervasive popularity in the scientific community is due to network simulations. Moreover, besides real-time simulations and network emulations, OMNeT++ has been extended to evaluate the performance of distributed and parallel systems [33]. Followings are the strengths of OMNeT++ that play an important role in its success and popularity in academia related to the simulation of network/distributed/parallel systems [4] [34]:

- OMNeT++ provides powerful Eclipse-based Integrated Development Environment (IDE).
- Interactive graphical runtime interface (Tkenv) for simulations that supports inspection, animation, and debugging to make visible the internals of a simulation.
- Availability of Command-line runtime interface (Cmdenv) for simulation execution.
- Easy to use NETwork Description (NED) language for systems' topology description.
- Simulation kernel library, which follows the rules and simple paradigm of well-known Object-Oriented C++ language.
- OMNeT++ is easy to use as it does not require extra coding for tracing and debugging capabilities.
- OMNeT++ is able to simulate all types of wired and wireless networks including Overlay [35], Peer-to-Peer (P2P) networks [36], Wireless Sensor Networks (WSNs) [37], Vehicular Ad-hoc Networks (VANETs) [38], Wireless Body Area Networks (WBANs) [39] etc.
- Available frameworks (INET [40], MiXiM [41], Oversim [42], Castalia [43] [44] etc.) provides standard model library for various protocols related to the layers of TCP/IP stack of wired and wireless networks.
- OMNeT++ provides an easy way to produce documentation.
- Availability of API documentation and a number of sample simulations.

Major components of OMNeT++ IDE are highlighted in Figure 3.1; however, the details related to OMNeT++ IDE components are available in [34, 45].

The OMNeT++ has been supported by major operating systems i.e., Windows 7 or later, Linux distributions, MAC OS X10.10 or later. Installation details on each operating systems are available in [46]. In addition, a number of OMNeT++ frameworks are available that provide a model for internet stack for various types of network. These main frameworks include:

- **INET Framework** provides an implementation of Internet stack models (i.e., IPv4/IPv6, UDP/TCP, and OSPF etc.), wired/wireless link layer protocols (i.e., Ethernet, 802.11 etc.). In addition, INET framework also provides support for various MANET routing protocols, mobility and DiffServ etc.
- **Veins** framework based on the integration of OMNeT++ and SUMO (a traffic simulator) is used for simulation inter-vehicle simulations.
- **INETMANET** is mainly used for MANETs.
- **OverSim** framework is used for the simulation of peer to peer networks.
- **SimuLTE** is used for the evaluation of Long-term Evolution cellular networks.
- **Castalia** framework considers low-power embedded devices and is used for WSNs and WBANs. Significant features of Castalia include implemented various MAC/routing protocols, physical channel modeling etc.
- **ANSAINET** allows analysis and configuration of real networks.
- **RINASim** stands for Recursive Internetwork Architecture.
3.2 Simulation Modeling in OMNeT++

In general, the OMNeT++ model comprises of two types of modules i.e., simple and compound modules. Both simple and compound instance modules types can be served as components of complex module types. Simple modules (independent or residing within a compound module) are active components that communicate through message passing over a channel while considering network descriptions and algorithm implementations with assigned configuration parameters [47]. Other components, which are required to make up an OMNeT++ simulation, include messages, channels, topology configuration files and algorithms as C++ functions [48]. To define a model for simulation in OMNeT++, the user is required to create a system of modules, which actually represents the system to be simulated over defined channels. Details of each component structure are discussed in section 3.3.

3.3 OMNeT++ Network Simulation Structure

Concerning the logical architecture of OMNeT++, Simple and Compound modules communicate with each other through message passing [3]. Message passing between modules is possible through connections. Connections exist in the form of gates, which are input/output module interfaces. Gates of simple and compound modules can be interconnected and generally a message pass through a connection chains. Considering this modular structure, the process of building an executable simulation program is shown in Figure 3.2. It becomes obvious that regarding any simulation project related to wired or wireless networks in OMNeT++, essentially following four types of files are required to build all scenarios (simple as well as complex):

- Network Description (NED) Language and files with (.ned) extension
- Algorithm implementation C++ File - files with (.cc) extension
- Configuration File – files with (.ini) extension
- Message Files with (.msg) extension

In addition, simulation kernel libraries with the command line (cmd_env) and graphical user interface (Tkenv) are also required for the successful implementation of OMNeT++ simulation. Related details of topology description, implementation, and configuration in relevant files (i.e., .ned, .cc, .ini) have been discussed in the following subsections.
3.3.1 NED Language and Structure of .ned files

NED stands for NEtwork Description and this language is designed to build network models in OMNeT++ [49] [50]. NED language is also known as flexible topology description language. NED language with its simple syntax is a powerful language to describe the network model. Significant features that let NED language a success are [34, 51]:

- Hierarchical nature and component-based nature of modules,
- Independence of module and channel interfaces
- Support of inheritance to make subclasses of modules and channels
- Java-similar package structure that is easy to learn and reduce name clashes
- Availability of metadata annotations
- Flexible topology description
- Support model partitioning for parallel execution

NED language (programmed using NED files with .ned extensions) is used to define the structure of modules and the topology of the network. At the most basic level, we can roughly divide the contents of NED file into two fundamental components i.e., modules’ definition part and network topology definition part. Modules are of two types i.e., simple (active) and compound, which eventually represent the structure of each node that would be the part of simulation analysis. Simple modules fundamentally express module interface (i.e., gates and parameters). Active modules are programmed in C++ and the hierarchical nesting of simple modules form a compound module. On the other hand, a group of active modules can be encapsulated to form compound modules where hierarchy levels are not limited. Compound modules typically contain submodule definitions and interconnection. Network part describes the topology/layout of any network scenario or emplacement of certain nodes within a simulation scenario. Other features of NED file include inheritance (for modules, channels etc.), metadata notations, and package information.

NED files can be viewed as both graphically or in text mode. Figure 3.3 shows an example of code snippet for a simple module named as Node in NED text editor with its representation in design mode. Figure 3.4 shows an example of code snippet for a compound module named as Node in NED text editor with its representation in design mode containing simple modules named as appl, transL, ipL, dlL, and phyL.
3.3.2 Algorithm Implementation and Structure of the .cc files

Implementation of modules and the required functionality of the network (i.e., Module/Node definition and Network Topology as defined in the corresponding .ned file), can be achieved by writing C++ code in files with .cc extension. C++ code snippet (following object-oriented paradigm) shown in Listing 3.1, which is related to the implementation of message passing between two nodes \((N1 \text{ and } N2)\) of type \textit{Node} (as shown in Figure 3.5). Figure 3.4 shows the mapping of NED design view modules with NED source view modules.

![Diagram of C++ code snippet for Node implementation](image)

Figure 3.3: Example of simple module in design and source view

```
simple Node
{
    parameters:
    // No display icon and color
    @display("i-device/pc", gold);
    // For Node input/output Gates
    gates:
    // Gate to receive Packet
    input in;
    // Gate to send Packet
    output out;
}
```

Figure 3.4: View (design and source) of a compound module
Listing 3.1: Implementation of Simple Module named as Node

```cpp
#include <string.h>
#include <omnetpp.h>
using namespace omnetpp;
class Node : public cSimpleModule{
    protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
}

Define_Module(Node);
```

Figure 3.5: NED file source view of network consisting of two nodes directly connected to each other

3.3.3 Configuration Options and Structure of the .ini files

Execution attempt after writing .ned and .cc files in an OMNeT++ project will produce an error regarding the absence of configuration file i.e., file with a .ini extension. omnetpp.ini is the key configuration that file provides an opportunity to execute the specified network within an OMNeT++
project (because several networks can exist in the same OMNeT++ project). In addition, the user can also pass values as parameters (defined in .ned) from omnetpp.ini file. Associated with a defined network of the .ned file (shown in Figure 3.5) a simple code snippet of the omnetpp.ini file is shown in Figure 3.6.

![omnetpp.ini file](image)

**Figure 3.6**: Source view of omnetpp.ini for MyNetwork project

In omnetpp.ini file, a number of configuration options can be mentioned to control the execution of the simulation. Few configuration options are listed below [51]:

- **sim-time-limit** and **CPU-time-limit** options used to mention limits that how long the simulation should run
- **simtime-resolution** option is used to define resolution for simulation time (i.e., in terms of SI units s, ms, us, ps, fs etc.) or in terms of power-of-ten multiples of such units (i.e., 100ms)
- **record-eventlog** option is used to record simulation events into a log file (i.e., elog file)
- **fingerprint** – In OMNeT++, while running simulation, the simulation kernel computes a checksum while considering module id and current simulation time of each event. The fingerprint option compares the computed checksum with already declared one and it is responsible for generating an error if there exist significant changes that alter the behavior of the simulation model.
- **realtimescheduler-scaling** option allows you to scale simulation execution.

**NOTE**: It is also possible to specify configuration options in separate .ini files (i.e., realtime.ini and fingerprint.ini etc.).

**Review Questions**

1. Which features of OMNeT++ distinguishes it from the other popular network simulators?
2. What are the basic components of OMNeT++ architecture?
3. What is the significance of NED files in logical architecture of OMNeT++? How NED language helps a network engineer to model a simulation?
4. Which functions are required to be overridden to implement a simulation scenario in OMNeT++? and when these overridden functions are invoked during a simulation.
5. Explain different configurations that can be mentioned in .ini file(s).
CHAPTER - 4

General Structure of OMNeT++ Project

Learning Objectives:

After studying this chapter, you will be able to describe:

- the OMNeT++ project
- basic steps to build an OMNeT++ project
- creation of module and network topology
- creation of implementation files to the corresponding module/network topology
- creation of simulation configuration files
4.1 The OMNeT++ Project

To understand the working and performance of a network system, in this chapter we create an OMNeT++ project is required to be built. A typical OMNeT++ project contains files related to module definition, topology information, and algorithmic definitions. Concerning modeling and simulation of a network system using OMNeT++, details in this chapter furnish the basis for the next chapters.

4.2 Building an OMNeT++ Project

This chapter is designed to demonstrate the building and working of a simple scenario of two directly connected computers involved in message-passing. To begin with, we present the project structure (containing subdirectories) in a generic way that reader should consider for the projects. Steps and details are given below:

1- Start OMNeT++ Academic Edition; choose New from the File menu.
2- Select New→OMNeT++ Project (as shown in Figure 4.1 (a))
3- In the New OMNeT++ Project window, enter the name of the project as “MyProj” (Figure 4.1 (b)), and select both checkboxes (Use default location and Support C++ Development if not already checked) and click Next button. The default location for the creation of OMNeT++ project files is “sample” directory of installed OMNeT++. In the case of simple module writing, it is important to select the option Support C++ Development. However, if simple modules are not part of your project, then you may unselect this checkbox and all C++ features will be disabled for this project. In this example, the simple module is required to be created; therefore, we select the option of Support C++ Development.

![Figure 4.1 (a)](image-url)
Following next screen related to Initial Contents, select Empty Project option and click Next (Figure 4.1 (c)). Selection of Empty project with ‘src’ and ‘simulations’ folders option will create (simulations and src) folders within the created project.

In C++ Project Type window (as shown in Figure 4.2 (a)), select C++ Project Type with Toolchains that are supported by available platform and press Next button. Usually, only one option is available and therefore no change is required on this page.

Next, in C++ Project window (Figure 4.2 (b)) regarding Configuration Selection, check both options (gcc-debug and gcc-release if not already selected) and then press the Finished button. The gcc-debug option (also known as unoptimized build) is required in case of debugging and the gcc-release option (also known as optimized build) is required for code optimization.

Next, MyProj folder will appear in Project Explorer window containing the subfolders Includes and package.ned (Figure 4.3 (a) and (b)). Includes subfolder contains all necessary header files related to namespace omnetpp and GNU ISO library, gcc etc. By default, package.ned contains information related to project folder(s) details and license. The default license declaration line is @license(LGPL). LGPL stands for Lesser General Public License and it describes that this package code can be copied and distributed without changing. This license line is optional and can be changed regarding the user need.
We need to create a subfolder to store the C++ implementation files. To create this subfolder (named CC Src), right click on MyProj folder and press Source Folder button (Figure 4.4 (a)). Enter CC Src in the Folder name text field and click Finish button (as shown in Figure 4.4 (b)). The CC Src subfolder will appear in MyProj folder within Project Explorer window (Figure 4.4 (c)).
9- We have to create another subfolder to store network description files. To create a subfolder (named NEDSrc) for .ned files, right click on MyProj folder and press Folder (Figure 4.5 (a)). Enter NEDSrc in the Folder name and click Finish (Figure 4.5 (b)). The NEDSrc subfolder will appear in MyProj folder within Project Explorer window (Figure 4.5 (c)). All project related NED (except configuration) files will be saved in this folder.

10- To create a network description file (with .ned extension), right click on the NEDSrc subfolder and then select New→Network Description File (NED) as shown in Figure 4.6 (a).

11- New NED File window will appear. Select NEDSrc subfolder, write MyNetwork.ned in File name textfield and click Next button (Figure 4.6 (b)).

12- Select Empty NED File click Next and then Finish to proceed (Figure 4.6 (c) and (d)). The MyNetwork.ned file will appear within NEDSrc folder in Project Explorer window (Figure 4.6 (e)). Other options are also available here e.g. NED file with one item, Random, Ring, Grid, Tree Topologies generations, and importing of a topology. Selecting NED file with one item will allow you to choose one option as initial content i.e., simple module/compound module, network, channel, module interface, and channel interface etc. Selection of Random Topology option will lead you to further options regarding a number of nodes, links, as well as other NED types regarding node and channel. Ring, Grid, Tree topology options will allow you to create a network using one of the available technologies. However, for this project, Empty NED File choice is required to be selected.
Figure 4.6: (a) Creation, (b) Naming of NED File, (c) Selection, (d) Completion of initial contents, and (e) Viewing of NED file in project explorer.

13- Double click on *MyNetwork.ned* file and write code (as shown in Listing 4.1) in the *Edit Window*.

14- NED Design view (shown in Figure 4.7) provides the graphical representation of source code for NED file.
Listing 4.1  NED source view for simple module (Node) and Point-to-Point network

Line 1: `package myproj.NEDSrc;
        // A simple module creation named Node having two gates
Line 2: `simple Node{
Line 3:   gates:
Line 4:     input inGate;
Line 5:     output outGate;
Line 6: }

        /* Network creation consisting of two nodes (N1 and N2 of type
Node) */
Line 7: `network Pt2Pt_Network{
Line 8:   submodules:
Line 9:     N1: Node;
Line 10:    N2: Node;
Line 11:  connections:
Line 12:     N1.outGate --> { delay = 100ms; } --> N2.inGate;
Line 13:     N1.inGate <-- { delay = 100ms; } <-- N2.outGate;
Line 14: }

Figure 4.7: Design view of NED file

Listing 4.1 (which represents the design view shown in Figure 4.7) explains the creation of a simple module and network topology within NED file. The implementation starts with `package statement identifying the package that simulation program belongs to (line 1 of Listing 4.1). Lines 2—6 provide implementation details of a simple module creation (called `Node) in NED file having one input and one output gate named as `inGate and `outGate, respectively. The `Submodules section of a Point-to-Point network (named as `Pt2Pt_Network) block explains the creation of two nodes (N1 and N2) of type `Node (lines 8—10). `Connections block of this Point-to-Point network depicts the direct link between the nodes N1 and N2, respectively (lines 11—13). Established connection between two nodes represents channel. Parameters can be assigned to channels, for example, delay, data rate etc. Moreover, display strings can be used with connections (i.e., `ls, `t, `tt, `m for `line style/color, `text, `tooltip, `orientation/positioning, respectively). Lines 12—13 of Listing 4.1 display delay parameter, which represents 100ms propagation delay on the bidirectional connection between N1 and N2 to deliver messages from both nodes to each other. In addition, display string (to show connection line in red with width 3) can be mentioned for connection as:
N1.outGate --> { @display("ls = red,3"); } --> N2.inGate;

Similarly, display string can be used to assign a string to a channel as:
N1.outGate --> { @display("t = 5Mbps"); } --> N2.inGate;

15- To create C++ source file (relevant to MyNetwork.ned) right click on the CCSrc subfolder and then select New→Source File (as shown in Figure 4.8 (a)).

16- New Source File window will appear (Figure 4.8 (b)), write Node.cc in Source file textfield, select Default C++ source template and click Finish.

17- The Node.cc file will appear within CCSrc folder in the Project Explorer window (Figure 4.8 (c)).

18- Double click on Node.cc file, write code (Listing 4.2) in Code Edit Window.
// Inclusion of directive files
Line 1: #include <string.h>
Line 2: #include <omnetpp.h>
// Importing of namespace
Line 3: using namespace omnetpp;

// Node class inheriting cSimpleModule class
Line 4: class Node : public cSimpleModule{
    // cSimpleModule class virtual function to be overridden
    protected:
    Line 6:     virtual void initialize() override;
    Line 7:     virtual void handleMessage(cMessage *msg) override;
    Line 8: };

    // To register Node class with OMNeT++
Line 9: Define_Module(Node);

    /* Initialize function invoked once only at the simulation
        beginning */
Line 10: void Node::initialize(){
        // N1 selection to send first message
Line 11:     if(strcmp("N1", getName()) == 0){
            // Creation/Sending of first message on output gate of node N1
Line 12:             cMessage *msg = new cMessage("MyMsg");
Line 13:             send(msg, "outGate");
Line 14:         }
Line 15:    }

    // Overridden handleMessage() method
Line 16: void Node::handleMessage(cMessage *msg){
Line 17:     send(msg, "outGate"); //Send message to output gate
Line 18: }

Listing 4.2 demonstrates the working implementation of a simple module and actual
message passing between two nodes (N1 and N2). The implementation starts with the inclusion
of two directives i.e., string.h and omnetpp.h (lines 1—2). The header file strings.h is required for the
usage of strcmp function in this project. The header file omnetpp.h is required to be included to
access all standard simulation class libraries of OMNeT++. Moreover, import the omnetpp
namespace to shorten names qualification such as omnetpp::prefix. Simple module
implementation starts with the declaration of Node class with two inherited virtual functions i.e.,
initialize() and handleMessage() of cSimpleModule class (lines 4—8). Registration of Node class
has to be done with OMNeT++ via the Define_Module() macro (line 9 of Listing 4.2), which must
be the part of implementation (.cc) files and can not be written in header (.h) files as compiler
generates code from it. The initialize() and handleMessage() functions are required to be
overridden to implement the desired functionality. At the beginning of this simulation, OMNeT++
creates two objects (N1 and N2) of class Node and initialize() function is invoked. The initialize() function
is invoked only once at the start of simulation for each simple module. To bootstrap N1-
N2-N1-N2 process, the node N1 has been selected to send the first message (lines 11—13).
Overridden handleMessage() method will be invoked whenever a node (either N1 or N2) in this
network will receive a message. Therefore, for continuous message passing between N1 and N2,
each node sends a message to output gate (as shown in lines 16—17). On the other side, each
node on packet reception continues the same process as available in the handleMessage() function.
19- To create a network initialization file (omnetpp.ini), right click on the MyProj folder and then select **New → Initialization File (.ini)** (as shown in Figure 4.9). This file is required to set configuration information such as simulation time, node parameter values etc.

20- Select MyProj folder and write file name as omnetpp.ini (if not already written) in **File name** text field and proceed with **Next** button (Figure 4.10 (a)).

21- **Empty Ini File** in **Select template** text area and press **Next** button (Figure 4.10 (b)). Press browse button to select network in the **New Ini File** window (Figure 4.10 (c)).

![Figure 4.9: Selection of initialization file (ini) from main menu](image)

22- From the available list, select NED type related to your project as shown in Figure 4.10 (d). The selected NED type will appear in the text field (as shown in Figure 4.10 (e)). Click **Finish** button to complete the creation of omnetpp.ini (Figure 4.10 (f)).

![Figure 4.10: (a) (b) (c) (d) corners](image)
Figure 4.10: (a) Naming, (b) Initial content selection, (c) ini file selection dialog box, (d) and (e) selection of NED type, and (f) completion dialog for omnetpp.ini

Note: Instead of adopting procedure from step 20 to step 22, you can complete the creation of omnetpp.ini as following:

(a) Select MyProj and write file name as omnetpp.ini (if not already written) in File name text field and click the Finish button.

(b) omnetpp.ini file will appear under MyProj in Project Explorer window with an error icon. Double click on this file in the Project Explorer window and in Edit window write:

```
network = myproj.NEDSrc.MyNetwork
```

(c) Right click the MyProj folder icon in Project Explorer window and Build Project. Error icon will disappear from omnetpp.ini and project is now ready to run.

Figure 4.11: View of the source editor

23- The project files can be seen in project explorer (Figure 4.12 (a)).

24- Right click on the MyProj folder icon in the Project Explorer window and Build Project. The building of project may take several minutes (as shown in Figure 4.12 (b)) and then the project will be ready to run. Several user interfaces are available in OMNeT++ to run a simulation for example (Tkevn GUI, Qtenv GUI, and cmdenv GUI for batch execution).

Figure 4.12: (a) Project files in project explorer (b) Build project window
25- Run the project using play button in the menu bar (Figure 4.13). Simulation Tekenv window will appear (Figure 4.14) and then execute the simulation using run button in the menu bar. Similarly, Run (history, configuration) and Debug (history, configuration) options are available under Run option in the main menu bar.

**Figure 4.13:** Executing a project

![Figure 4.13: Executing a project](image)

**Figure 4.14:** Project execution (message passing between two nodes N1 and N2)

![Figure 4.14: Project execution](image)

**Practice Tasks**

1. Create a network consisting of two nodes: PC1 and PC2. Node PC1 should generate and send packets (with incremental packet IDs i.e., Pk1, Pk2, Pk3,...) to PC2 node. At node PC2, it should send back numbered acknowledgement (i.e., Ack1, Ack2, Ack3,...) messages to node PC1. Hint: To generate a message/packet having different sequence number each time, following code snippet will be helpful:

   ```
   char msgname[20];
   sprintf(msgname, "tic-%d", ++seq);
   cMessage *msg = new cMessage(msgname);
   ```

2. Create a network consisting of three nodes i.e., Server, PC1, and PC2 (Topology shown in Figure 4.15). Both nodes PC1 and PC2 should generate and send a message to the Server. On receiving a packet, the Server should send back the same packet to the sending
node. Similarly, whenever a node receives a message from a server, it should send back that message to the server within a time delay of 500 ms.

Figure 4.15: Network topology for practice task 2
CHAPTER – 5

Implementation of Layered Architecture

Learning Objectives:

After studying this chapter, you will be able to:

- explain the concept of layers in TCP/IP
- implement layered architecture modules in OMNeT++
- understand the connectivity of TCP/IP layers in OMNeT++
- explain the mechanism of data flow through TCP/IP layers
5.1 Overview
Concerning software implementation, it is already explained that general blueprint developed by network designers (also known as Network Architecture) ultimately deals with the complexities associated with the network applications. Network designers consider abstractions that generally lead to layering that ultimately provides two fundamental features i.e., component-level decomposition and design modularity [6]. On the other hand, related to simulation analysis, it was also mentioned in the introductory chapter (of this book) that an OMNeT++ model basically consists of message passing communication modules (simple as well as compound modules). The compound modules are typically a representation of both end system (i.e., clients, servers, peers, etc.) and core (switch, router, bridge, gateways) network devices. However, simple modules (independent or residing within the compound module) are active components that communicate through message passing over a channel while considering the network descriptions and algorithm implementations with assigned configuration parameters [52]. Implementation of the simple module has already been described in Chapter 4. Concerning the availability of a compound module of OMNeT++, this chapter is designed to demonstrate the building blocks of the layered architecture of a computer network.

5.2 TCP/IP Architecture
The Transmission Control Protocol/Internet Protocol (TCP/IP) [53, 54] suite also known as Internet Architecture was developed by Internet Engineering Task Force (IETF). The TCP/IP is modeled in the five layers and supports a number of standards and non-standard applications through a set of core protocols at each layer. Concerning the functionality of each layer of TCP/IP suite, a brief discussion has been already provided in Section 1.2. This section is related to the brief discussion of three important features of TCP/IP suite. The first feature of TCP/IP suite is that it does not imply strict layering. This feature facilitates application programmers to bypass any defined layer(s) for building new applications (that can run on top of any existing protocol of any layer). The second feature of TCP/IP is that the IP serves as the focal point in this architecture. To support packet exchange using different types of transport technologies, IP layer of TCP/IP suite provides common methods. Therefore, IP layer clearly separates the issues of host-to-host and process-to-process communication. The third characteristic of TCP/IP architecture follows IETF culture of developing new protocols such as implementation before standardization, which actually ensures efficient implementation.

5.3 Layered Architecture in OMNeT++
In this chapter, simulation is designed to demonstrate the implementation of the layered architecture of a network using the availability of compound modules of OMNeT++. Two end systems (i.e., Host A and Host B) in OMNeT++ can be considered as a compound module, which are involved in communication with each other. However, the layers of TCP/IP stack (i.e., Physical, Data Link, Network, Transport, and Application layers) in Host A and Host B are simple modules connected to each other through gates as shown in Figure 5.1.

5.4 Building a Layered Architecture in OMNeT++
In this section, we will design a simulation using the layered architecture based on TCP/IP protocol stack. We will implement each layer connections with corresponding adjacent layers. Moreover, we will simulate the mechanism of a point-to-point connection using message passing. To create a simulation of Point-to-Point network with compound module consisting of five simple modules (symbolizing the five layers of the TCP/IP stack), description of each step is given below:

1- To create TCP/IPStack project, follow the steps (1—9) of section 4.2.
2- To create a network description file (with .ned extension), right click on the NEDSrc subfolder and then select New→Network Description File (NED) (as shown in Figure 5.2 (a)).
3- New NED File window will appear. Select NEDSrc, write PhysicalLayer.ned in File name textfield and click Next button (Figure 5.2(b)).
4- Select *Empty NED File*, click Next button and then click *Finish* to proceed (as shown in Figure 5.2(c) and Figure 5.2(d)). The *PhysicalLayer.ned* will appear within *NEDSrc* folder in *Project Explorer* window (Figure 5.2(e)).

![Figure 5.1: Simple or compound modules representing layers with Input (I) / Output (O) gate pairs in OMNeT++](image)

(a)

(b)

(c)
5- Repeat steps 3—4 to create other four NED files such as `DataLinkLayer.ned`, `NetworkLayer.ned`, `TransportLayer.ned`, and `ApplicationLayer.ned`. All created `.ned` files will appear within `NEDSrc` folder in the `Project Explorer` window (as shown in Figure 5.3).

6- Write the following code in the `ApplicationLayer.ned` file and save.

```
Listing 5.1  Source view of ApplicationLayer NED

Line 1: package tcpipstack1.NEDSrc;
Line 2: simple ApplicationLayer{
Line 3:   gates:
Line 4:     inout appGate[1];
Line 5: }
```

Listing 5.1 demonstrates the creation of simple module as Application layer with one gate (appGate). The type of appGate is `inout` (line 4 of Listing 5.1), which allows bidirectional connections. Simulation kernel represents an `inout` gate as a gate pair (input/output). The Input and output gates of ApplicationLayer module are attached to the TransportLayer module. By adding the suffix (`$i` and `$o` for input and output, respectively) in implementation (.cc) file, both
sub-gates within a gate pair can be connected individually. Moreover, inout gates are connected to each other through double-head arrow (i.e., `-->` as shown in Listing 5.6).

7- Write the following code in the `TransportLayer.ned` file and save.

<table>
<thead>
<tr>
<th>Listing 5.2</th>
<th>Source view of TransportLayer NED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td><code>package tcpipstack1.NEDSrc;</code></td>
</tr>
<tr>
<td>Line 2:</td>
<td><code>simple TransportLayer{</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>parameters:</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>int transPortNo = 20;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>gates:</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>inout tcpGate[2];</code></td>
</tr>
<tr>
<td>Line 7:</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Listing 5.2 provides the creation of simple module representing the Transport layer with two gate pairs (line 6). One pair of input/output gate is linked with the `ApplicationLayer` module and the second inout gate pair is attached with `NetworkLayer` module. `TransportLayer` module has one parameter of type integer representing the Transport layer port number (line 4).

8- Write the following code in the `NetworkLayer.ned` file and save.

<table>
<thead>
<tr>
<th>Listing 5.3</th>
<th>Source view of NetworkLayer NED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td><code>package tcpipstack1.NEDSrc;</code></td>
</tr>
<tr>
<td>Line 2:</td>
<td><code>simple NetworkLayer{</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>parameters:</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>string ipAddress = &quot;NIL&quot;;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>gates:</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>inout ipGate[2];</code></td>
</tr>
<tr>
<td>Line 7:</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Listing 5.3 displays the creation of simple module representing the Network layer with two gate pairs (lines 1—7). One pair of input/output gate is linked with `TransportLayer` module and the second inout gate pair is attached with `DataLinkLayer` module. `NetworkLayer` module has one parameter of type string representing the IP address of the node (line 4).

9- Write the following code in the `DataLinkLayer.ned` file and save.

<table>
<thead>
<tr>
<th>Listing 5.4</th>
<th>Source view of DataLinkLayer NED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td><code>package tcpipstack1.NEDSrc;</code></td>
</tr>
<tr>
<td>Line 2:</td>
<td><code>simple DataLinkLayer{</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>parameters:</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>string macAddress = &quot;NIL&quot;;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>gates:</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>inout dllGate[2];</code></td>
</tr>
<tr>
<td>Line 7:</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Listing 5.4 explains the creation of simple module representing the Data Link layer with two gate pairs (line 1—7). One pair of input/output gate is linked with `NetworkLayer` module and the second inout gate pair is attached with `PhysicalLayer` module. `DataLinkLayer` module has one parameter of type string representing the MAC address of the node (line 4).

10- Write the following code in the `PhysicalLayer.ned` file and save.
Listing 5.5  Source view of PhysicalLayer NED

Line 1: `package tcpipstack1.NEDSrc;`
Line 2: `simplee PhysicalLayer{
Line 3:   gates:
Line 4:     inout phyGate[2];
Line 5: }`

Listing 5.5 describes the creation of simple module representing of the Physical layer with two gate pairs (lines 1-5). One gate pair is linked with DataLinkLayer module and the second inout pair with the output port of PC1 (shown in Listing 5.6 lines 26—27).

11- To create a compound module (with .ned extension), right click on the NEDSrc subfolder and then select New→Compound Module (as shown in Figure 5.4(a)).

![New Compound Module](image)

![New Compound Module](image)

![New Compound Module](image)

![New Compound Module](image)

Figure 5.4: (a) Creation, (b) Naming of compound module, (c) (d) (e) Selection of initial contents, and viewing of NED file in project explorer window

12- New Compound Module window will appear. Select NEDSrc, write PC.ned in File name textfield and click Next button (Figure 5.4(b)).
13- Select **An Empty Compound Module**, click **Next** and then **Finish** to proceed (Figure 5.4(c) and (d)). **PC.ned** will appear within **NEDSrc** folder in **Project Explorer** window (Figure 5.4 (e)).

14- Write the following code in the **PC.ned** file and save

**Listing 5.6 Implementation of PC module and network description in NED**

```
// Package statement
Line 1: package tcpipstack1.NEDSrc;
// A compound module creation named PC having two gates
Line 2: module PC{
Line 3:   parameters:
Line 4:     @display("i=device/pc2,gold");
Line 5:   gates:
Line 6:     inout gate[];
     // Inclusion of Layers as submodules in PC module
Line 7:     submodules:
Line 8:       appL: ApplicationLayer {
Line 9:         @display("p=45,26;i=block/sink");
Line 10:     }
Line 11:       tcpL: TransportLayer {
Line 12:         @display("p=123,56;i=block/filter");
Line 13:     }
Line 14:       ipL: NetworkLayer {
Line 15:         @display("p=123,137;i=old/proc2");
Line 16:     }
Line 17:       dlL: DataLinkLayer {
Line 18:         @display("p=123,216;i=device/card");
Line 19:     }
Line 20:     phyL: PhysicalLayer {
Line 21:         @display("p=57,226;i=device/card");
Line 22: }
Line 23:     connections:
Line 24:      appL.appGate[0] <-> tcpL.tcpGate[0];
Line 25:      tcpL.tcpGate[1] <-> ipL.ipGate[0];
Line 26:      ipL.ipGate[1] <-> dlL.dllGate[0];
Line 27:      dlL.dllGate[1] <-> phyL.phyGate[0];
Line 28:      phyL.phyGate[1] <-> gate++;
Line 29: }
```

Listing 5.6 displays the creation of a compound module named **PC** in the NED file. The implementation starts with **package statement** (line 1) indicating respective package to be included. Representing layers of TCP/IP stack, lines 8—22 provide the inclusion of already created simple modules (i.e., ApplicationLayer, TransportLayer, NetworkLayer, DataLinkLayer, and PhysicalLayer). As submodules in a compound module do not require gate vector to be specified; therefore, it has been left empty. The **gate++** operator automatically expands the vector. @display property of module and submodules has been used to specify icon, color, and position of respective module and submodules.

15- To create your own packet (with **.msg** extension), right click on Project (**TCP/IPStack**) folder and then select **New→Message Definition (msg)** (as shown in Figure 5.5 (a)).

16- **New Message File** window will appear. Select **TCP/IPStack**, write **packet.msg** in **File name**textfield and click **Next** button (Figure 5.5(b)).

17- Select **Empty Message File** as **Initial Contents** and click **Next** and then **Finish** to proceed (Figure 5.5 (c) and (d)). **packet.msg** file will appear within **TCP/IPStack** folder in **Project Explorer** window (Figure 5.5 (e)).

45
Figure 5.5: (a) Creation, (b) Naming of packet, (c) (d) (e) Selection of initial contents, and viewing of message file in the project explorer window.

Attributes (fields) of packet header are shown in Listing 5.7.

```
Listing 5.7 Attributes of packet

Line 1:  packet Packet{
Line 2:   int srcAdd = 0;
Line 3:   int destAdd = 0;
Line 4:   int appHeader = NULL;
Line 5:   int tcpHeader = NULL;
Line 6:   int ipHeader = NULL;
Line 7:   int dllHeader = NULL;
Line 8:   int phyHeader = NULL;
Line 9: }
```
Select **Project** from the menu bar and click **Build Project** (as shown in Figure 5.6). On the completion of the building process, two packet implementation files named as **packet_m.h** and **packet_m.cc** (Figure 5.7) will be created. These two files (header file and C++ file) are actual files of packet implementation with required getter/setter functions of packet attributes. Besides getter/setter functions of packet attributes, packet implementation files contains `dup()`, `parsimPack()`, and `parsimUnpack()` functions. The `dup()` function is required to create duplicate packet copies and `parsimPack()` or `parsimUnpack()` functions are needed for parallel simulation. Moreover, it is custom to provide the copy constructor and assignment operator to invoke `copy()` function to copy local members of packet class.

**Figure 5.6:** Menu selection to build project

**Figure 5.7:** View of header file with C++ implementation file in project explorer window

To create C++ source files (relevant to **PhysicalLayer.ned**, **DataLinkLayer.ned**, **NetworkLayer.ned**, **TransportLayer.ned**, **ApplicationLayer.ned**), right click on the **CCSrc** subfolder and then select **New→Source File** (as shown in Figure 5.8(a)).

**New Source File** window will appear, write **PhysicalLayer.cc** in **Source file** textfield, select **Default C++ source template** and click **Finish** (Figure 5.8(b)). **PhysicalLayer.cc** file will appear within **CCSrc** folder in the **Project Explorer** window (Figure 5.8(c)).

Repeat steps 20—21 to create other 4 source files i.e., **DataLinkLayer.cc**, **NetworkLayer.cc**, **TransportLayer.cc**, and **ApplicationLayer.cc**. All created .cc files will appear within **CCSrc** folder in **Project Explorer** window (Figure 5.9).
Figure 5.8: (a) Creation, (b) Naming, and (c) View of CC file

Figure 5.9: CC files in project explorer window
The implementation of the Application layer (as shown in Listing 5.8) starts with the inclusion of directives. Here, packet_m.h file is required to be included to access all getter/setter functions regarding attributes (i.e., srcAdd, destAdd, data, appHeader, tcpHeader,
ipHeader, dllHeader, phyHeader) of class Packet, which is relevant to created packet.msg. At the beginning of this simulation, OMNeT++ creates two objects (PC1 and PC2) of class PC and initialize() function is invoked. To bootstrap, the Application layer of module PC2 (having id=9 in our simulation) has been selected to send a first packet (with the inclusion of string [AppHeader]) shown in lines 12—18. You may provide Application layer id of module PC1 to start message passing. The scheduleAt(simtime_t t, cMessage *msg) function schedules a self-message. It means that message will be delivered back to the same module via receive() or handleMessage() at simulation time t. Moreover, using this method, you can implement timers. Upon receiving the packet, handleMessage() function will be invoked. Here, implementation of handleMessage() function shows that it performs type casting of received packet into the created packet (line 20 in Listing 5.8). If handleMessage() function is invoked on received packet (line 25 in Listing 5.8), it shows all packet header details and sends back the packet to output gate of the Application layer. But, in the case of sending a newly created packet (line 21 in Listing 5.8), it will pop (bubble) notification about the addition of new application header and send a packet to the output gate of application header. The EV macro is required for logging purpose.

24- Write following C++ code for TransportLayer.cc.

<table>
<thead>
<tr>
<th>Listing 5.9</th>
<th>Implementation of TransportLayer Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td><code>#include &lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td>Line 2:</td>
<td><code>#include &lt;string.h&gt;</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>#include &lt;omnetpp.h&gt;</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>#include &lt;packet_m.h&gt;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>#include &lt;vector&gt;</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>using namespace omnetpp;</code></td>
</tr>
<tr>
<td>Line 7:</td>
<td><code>class TransportLayer : public cSimpleModule{</code></td>
</tr>
<tr>
<td>Line 8:</td>
<td><code>private:</code></td>
</tr>
<tr>
<td>Line 9:</td>
<td><code>cPar *transPortNo;</code></td>
</tr>
<tr>
<td>Line 10:</td>
<td><code>protected:</code></td>
</tr>
<tr>
<td>Line 11:</td>
<td><code>virtual void initialize() override;</code></td>
</tr>
<tr>
<td>Line 12:</td>
<td><code>virtual void handleMessage(cMessage *msg) override;</code></td>
</tr>
<tr>
<td>Line 13:</td>
<td><code>};</code></td>
</tr>
<tr>
<td>Line 14:</td>
<td><code>Define_Module(TransportLayer);</code></td>
</tr>
<tr>
<td>Line 15:</td>
<td><code>void TransportLayer::initialize(){</code></td>
</tr>
<tr>
<td>Line 16:</td>
<td><code>//Acquire TCP Port No. declared in TransportLayer.ned</code></td>
</tr>
<tr>
<td>Line 17:</td>
<td><code>transPortNo = &amp;par(&quot;transPortNo&quot;);</code></td>
</tr>
<tr>
<td>Line 18:</td>
<td><code>};</code></td>
</tr>
<tr>
<td>Line 19:</td>
<td><code>void TransportLayer::handleMessage(cMessage *msg){</code></td>
</tr>
<tr>
<td>Line 20:</td>
<td><code>//Conversion and Concatenation to complete TCP Header</code></td>
</tr>
<tr>
<td>Line 21:</td>
<td><code>long portno = transPortNo-&gt;longValue();</code></td>
</tr>
<tr>
<td>Line 22:</td>
<td><code>string hs = &quot;[TCPHeader]&quot;;</code></td>
</tr>
<tr>
<td>Line 23:</td>
<td><code>string ctcpH = hs + to_string (x);</code></td>
</tr>
<tr>
<td>Line 24:</td>
<td><code>const char* transH = tpn.c_str();</code></td>
</tr>
<tr>
<td>Line 25:</td>
<td><code>//For packets to be received from Application Layer</code></td>
</tr>
<tr>
<td>Line 26:</td>
<td><code>if(pkt-&gt;arrivedOn(&quot;tcpGate$i&quot;,0)){</code></td>
</tr>
<tr>
<td>Line 27:</td>
<td><code>pkt-&gt;setTcpHeader(transH);</code></td>
</tr>
<tr>
<td>Line 28:</td>
<td><code>bubble(&quot;[TcpHeader ADDED]&quot;);</code></td>
</tr>
</tbody>
</table>
Line 23:    EV<<"Message contains"<<pkt->getAppHeader()  
          <<pkt->getTcpHeader()<<pkt->getIpHeader()  
          <<pkt->getDllHeader()<<pkt->getPhyHeader()<<endl;  
Line 24:    send(pkt, "tcpGate$o",1);  
    }  
    //For packets to be received from Network Layer  
Line 25:    if(pkt->arrivedOn("tcpGate$i",1)){  
    Line 26:        pkt->setTcpHeader(NULL);  
    Line 27:        bubble("[TcpHeader REMOVED]");  
    Line 28:        EV << "Message contains"<<pkt->getAppHeader()  
          <<pkt->getTcpHeader()  
          <<pkt->getIpHeader()  
          <<pkt->getDllHeader()  
          <<pkt->getPhyHeader()<<endl;  
    Line 29:        send(pkt, "tcpGate$o",0);  
    Line 30:    }  
    Line 31: }  

Using par() member function (line 16 of Listing 5.9) at initialization phase, the Transport layer  
takes the value of module parameter (transPortNo) declared in the NED files. NED declared  
module parameters are represented with cPar class at runtime and accessed through par()  
member function. Upon receiving a packet, handleMessage() function will be invoked. Here,  
implementation of handleMessage() function shows that it performs type casting of received  
packet into the created packet (line 19). Next lines of code show that if the packet received from  
the Application layer (line 20), the TCPHeader will be set and forward this packet to the Network  
layer. However, in the case of reception of a packet from Network layer (line 25), the TCPHeader  
will be set to null and forward this packet to the Application layer.

25- Write following C++ code for NetworkLayer.cc.

<table>
<thead>
<tr>
<th>Listing 5.10</th>
<th>Implementation of NetworkLayer module</th>
</tr>
</thead>
</table>
| Line 1: #include <stdio.h> | Line 7: class NetworkLayer : public cSimpleModule{
| Line 2: #include <string.h> | Line 8:     protected:
| Line 3: #include <omnetpp.h> | Line 9:     virtual void initialize() override;
| Line 4: #include <packet_m.h> | Line 10:    virtual void handleMessage(cMessage *msg) override;
| Line 5: #include <vector> | Line 11: };
| Line 6: using namespace omnetpp; | Line 12: Define_Module(NetworkLayer);
| Line 13: void NetworkLayer::initialize(){} | Line 14: void NetworkLayer::handleMessage(cMessage *msg){
| | Line 15:     char* ipH = "[IpHeader]";
| | Line 16:     Packet *pkt = check_and_cast<Packet *>(msg);  
| | Line 17:     if (pkt->arrivedOn("IpGate$i",0))
| |             pkt->setIpHeader(ipH);
| | Line 18:         bubble("[IpHeader ADDED]");  
| | Line 19:         EV <<"Message contains"<<pkt->getAppHeader()  
| | Line 20:     pkt->setIpHeader(NULL);
| | Line 21: }  
| | Line 22: }  
| | Line 23: void NetworkLayer::receiveCp(tcpGate$o, Packet *pkt){  
| | Line 24:     if(pkt->arrivedOn("tcpGate$i",1))  
| | Line 25:         pkt->setTcpHeader(NULL);
| | Line 26:         bubble("[TcpHeader REMOVED]");  
| | Line 27:         EV << "Message contains"<<pkt->getAppHeader()  
| | Line 28:             <<pkt->getTcpHeader()  
| | Line 29:             <<pkt->getIpHeader()  
| | Line 30:             <<pkt->getDllHeader()  
| | Line 31:             <<pkt->getPhyHeader()<<endl;  
| | Line 32:     send(pkt, "tcpGate$o",0);  
| | Line 33: }  
| | Line 34: }  
| | Line 35: Define_Module(NetworkLayer);

51
Implementation of the Network layer is similar to that of the Transport layer with the difference that if the packet received from the Transport layer (line 17), IP header will be set and then forward this packet to the Data Link layer. Moreover, in the case of reception of a packet from the Data Link layer (line 22), the IP header will be set to null and forward this packet to the Transport layer.

26- Write following C++ code for DataLinkLayer.cc.

Listing 5.11  Implementation of DataLinkLayer module

```cpp
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include <packet_m.h>
using namespace omnetpp;

class DataLinkLayer : public cSimpleModule{
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(DataLinkLayer);

void DataLinkLayer::initialize() {
  // Initialize DataLinkLayer
}

void DataLinkLayer::handleMessage(cMessage *msg){
  char* dllH = "[DllHeader]";
  Packet *pkt = check_and_cast<Packet *>(msg);
  if(pkt->arrivedOn("dllGate$i",0)){
    pkt->setDllHeader(dllH);
    bubble("[DllHeader ADDED]");
    EV <<"Message contains"<<pkt->getAppHeader() <<pkt->getTcpHeader() <<pkt->getIpHeader() <<pkt->getDllHeader() <<pkt->getPhyHeader() << endl;
    send(pkt, "dllGate$o",1);
  }
}
```

Line 20: send(pkt, "ipGate$o",1);
Line 21: }
Line 22: if(pkt->arrivedOn("ipGate$i",1){
Line 23: pkt->setIpHeader(NULL);
Line 24: bubble("[IpHeader REMOVED]");
Line 25: EV <<"Message contains" <<pkt->getAppHeader() <<pkt->getTcpHeader() <<pkt->getIpHeader() <<pkt->getDllHeader() <<pkt->getPhyHeader() << endl;
Line 26: send(pkt, "ipGate$o",0);
Line 27: }
Line 28: }

Implementation of DataLinkLayer is similar to that of the Transport layer with the difference that if the packet received from the Transport layer (line 17), IP header will be set and then forward this packet to the Data Link layer. Moreover, in the case of reception of a packet from the Data Link layer (line 22), the IP header will be set to null and forward this packet to the Transport layer.

```cpp
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include <packet_m.h>
using namespace omnetpp;

class DataLinkLayer : public cSimpleModule{
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(DataLinkLayer);

void DataLinkLayer::initialize() {
  // Initialize DataLinkLayer
}

void DataLinkLayer::handleMessage(cMessage *msg){
  char* dllH = "[DllHeader]";
  Packet *pkt = check_and_cast<Packet *>(msg);
  if(pkt->arrivedOn("dllGate$i",0)){
    pkt->setDllHeader(dllH);
    bubble("[DllHeader ADDED]");
    EV <<"Message contains"<<pkt->getAppHeader() <<pkt->getTcpHeader() <<pkt->getIpHeader() <<pkt->getDllHeader() <<pkt->getPhyHeader() << endl;
    send(pkt, "dllGate$o",1);
  }
}
```
Implementation of Data Link layer is similar to that of the Network layer with the difference that if
the packet received from the Network layer (line 16), DLLHeader will be set and forward this packet
to the Physical layer. Moreover, in the case of reception of a packet from the Physical layer (line
22), the DLLHeader will be set to null and forward this packet to the Network layer.

27- Write following C++ code for PhysicalLayer.cc file.

### Listing 5.12 Implementation of PhysicalLayer module

```cpp
Line 1: #include <stdio.h>
Line 2: #include <string.h>
Line 3: #include <omnetpp.h>
Line 4: #include <packet_m.h>
Line 5: using namespace omnetpp;

Line 6: class PhysicalLayer : public cSimpleModule{
Line 7:   protected:
Line 8:   virtual void initialize() override;
Line 9:   virtual void handleMessage(cMessage *msg) override;
Line 10:};

Line 11: Define_Module(PhysicalLayer);

Line 12: void PhysicalLayer::initialize(){
Line 13:   void PhysicalLayer::handleMessage(cMessage *msg){
Line 14:     char* phyhddr="[PhyHeader]";
Line 15:     Packet *pkt= check_and_cast<Packet *>(msg);

Line 16:     if(pkt->arrivedOn("phyGate$i",0)){
Line 17:       pkt->setPhyHeader(phyhddr);
Line 18:       bubble("[PhyHeader ADDED]" );
Line 19:       EV << "Message contain"<<pkt->getAppHeader()
Line 20:         <<pkt->getTcpHeader()
Line 21:         <<pkt->getIpHeader()
Line 22:         <<pkt->getDllHeader()
Line 23:         <<pkt->getPhyHeader() << endl;
Line 24:       send(pkt, "phyGate$o",1);
Line 25:     }else
Line 26:     if(pkt->arrivedOn("phyGate$i",1)){
Line 27:       pkt->setPhyHeader(NULL);
Line 28:       bubble("[PhyHeader REMOVED]" );
```

53
Implementation of the Physical layer shows that upon receiving a packet from the Data Link layer (line 16), PhyHeader will be set and forward this packet to the Physical layer of other connected module of type PC. Moreover, in the case of reception of a packet from the Physical layer (line 21), the PhyHeader will be set to null and forward this packet to the Data Link layer.

To create a network description file (with .ned extension), right click on the NEDSrc subfolder and then select New→Network Description File (NED) as shown in Figure 5.10(a).

New NED File window will appear. Select NEDSrc, write Pt2Pt_Network.ned in File name textfield and click Next button (Figure 5.10 (b)).

Select Empty NED File click Next and then Finish to proceed (Figure 5.10 (c) and Figure 5.10(d)). Pt2Pt_Network.ned will appear within NEDSrc folder in Project Explorer window (shown in Figure 5.10 (e)).
Write the following code in the `Pt2Pt_Network.ned` file and save.

**Listing 5.13** Implementation of Point-to-Point network

```plaintext
| Line 1: | package tcpipstack1.NEDSrc; |
| Line 2: | network Pt2Pt_Network{ |
| Line 3: | submodules: |
| Line 4: | PC1: PC; |
| Line 5: | PC2: PC; |
| Line 6: | connections: |
| Line 7: | PC1.gate++ <-> PC2.gate++; |
| Line 8: | }
```

Listing 5.13 describes the creation of two nodes (PC1 and PC2) of type PC (lines 3—5) and use of gate++ operator (line 7) to automatically expand the empty gate vector.

To create a network initialization file (`omnetpp.ini`), right click on the `TCP/IPStack` folder and then select `New → Initialization File (.ini)` as shown in Figure 5.11.
Figure 5.11: Selection of initialization file (ini file) from main menu

33- Select **TCP/IP Stack** and write file name as **omnetpp.ini** (if not already written) in **File name** text field and proceed with **Next** button (Figure 5.12 (a)).

34- **Empty Ini File** in **Select template** text area and press **Next** (Figure 5.12 (b)).

35- Press browse button to select network in **New Ini File** window (Figure 5.12(c)).

36- From available list, select NED type related to your project as shown in Figure 5.16 (d). The selected NED type will appear in the text field (Figure 5.12 (e)). Click **Finish** to complete the creation of **omnetpp.ini** (Figure 5.12 (f)).
37- The project files can be seen in Project Explorer (Figure 5.13).

38- Right click on the TCPIPStack folder icon in Project Explorer window and Build Project. The building of project may take several minutes (Figure 5.14) and then the project will be ready to run.

39- Run the project using play button in the menu bar (Figure 5.15). Simulation Tekenv window will appear (as shown in Figure 5.16) and then Run the simulation using run button in the menu bar. Message Passing in executing simulation shown in Figure 5.17—5.18.
Figure 5.15: Execute project

Figure 5.16: Running project of OMNeT++

Figure 5.17: (a) (b) (c) Message passing through layers of PC1
We have simulated TCP/IP-based message passing between each layer for point to point network. On each stage, each layer added and removed headers during transmission and reception respectively. When you will run the simulation, you will observe the movement of messages through each layer.

**Practice Tasks**

1. Create a network with three nodes as shown in Figure 5.19. PC1 and PC2 are required to send a message to each other (passing through an intermediate node R). Intermediate node R will receive packets from both the sending nodes and forwards the received packet to the destination node. PC1 and PC2 should implement 1—5 TCP/IP layers while the intermediate node R implements only 1—3 TCP/IP layers.

2. Create a network having three nodes as shown in Figure 5.19. PC1 sends request (REQ) packet to R. R process the REQ packet and sends back reply (REP) packet to PC1. On receiving REP, PC1 will send a data packet with sequence number 1 (DATA1). Upon receiving DATA1, R will forward it to PC2 using the same REQ-REP mechanism. Similarly, transfer a packet from PC2 to PC1 using REQ-REP mechanism.
Implementation of Cyclic Redundancy Check (CRC)

Learning Objectives:

After studying this chapter, you will be able to describe:

- the basics of error detection
- the fundamentals of CRC
- implementation of CRC using OMNeT++
6.1 Error Detection

Data may get corrupted during transmission due to the several reasons such as noise, cross-talk etc. as the transmission medium is error-prone. The upper layers (i.e., Network, Transport, and Application) are not aware of actual hardware data processing and they assume error-free transmission between the systems. The magnitude of errors may span from one bit to several bits (bit and burst levels) that are introduced into frames. One of the major responsibilities of the Data Link Layer is to convert the unreliable error-prone physical link into a reliable communication pathway through error detection. The fundamental goal of error detection schemes is to achieve high error detection rate at low overhead (i.e., k << n where k and n represent redundant frame bits and original frame bits respectively). Errors are detected by means of Parity Check and Cyclic Redundancy Check (CRC). Both schemes involve the similar mechanism of padding a few extra bits with actual data to confirm that padded bits are received at other correctly. If the counter-check fails at the receiver side, the transmitted frames are considered corrupted. Parity check scheme is better in terms of overhead but is inefficient to detect errors at high burst error rate. On the other hand, Cyclic Redundancy Check CRC is efficient both in terms of error detection and low overhead due to less redundant bits [55].

6.2 Cyclic Redundancy Check (CRC)

In 1961, W. Wesley Peterson proposed the idea of CRC mechanism [56]. A Cyclic Redundancy Check (CRC) is one of the most used techniques to make communication reliable over the data-link layer.

In the digital networks, CRC is most widely used error detection code. Using the CRC, sudden changes in raw data during transmissions can easily be detected. The basic mechanism used is to attach additional bits (called check-value) to the raw data. The check-value is calculated using the remainder of a polynomial division of data contents. At the destination device, the process is repeated to re-calculate the CRC code and in the case of no match, the error is reported for appropriate actions of packet re-transmission.

The name CRC refers to the fact that additional (redundant) value i.e., check-value is appended to the message. The CRC mechanism is based on cyclic codes. Cyclic codes are the simple easy-to-implement special type of linear block-code that is useful for detecting common errors in binary data. To generate new code, circular shift mechanism is applied to the data bits; and this produces another binary word representing a new code.

An n-bit CRC means that its check value is n-bits long. For a given size of n, multiple CRC codes can be calculated each incorporating a different polynomial. The employed polynomial will have the highest degree n based on n + 1 terms (polynomial length equals to n+1). During transmission, a 16- or 32-bit polynomial is applied to a block of binary data being transmitted and the corresponding CRC code is calculated. This CRC code or check-value is appended to the message being transferred. At the receiving end, the same polynomial is applied to the data-block and the resultant check-value is then compared with the transmitted check-value. If both of the check-values are same this means that data was received without any error. In the case of different check values, the sender is requested to resend the data. Commonly, CRC-16 version is used that employs a 16-bit polynomial. A CRC-16 detects most of the single and double-bit errors and ensures high degree (up to 99.99%) of error detection. CRC-16 is generally used for data blocks up to 04 Kilobytes; larger data blocks require 32-bit CRC mechanism. The Ethernet and token ring protocols both employ 32-bit CRC. For more details about the use of polynomial in CRC, readers are recommended to consult section 2.4.3 in [6].

6.3 Implementation of CRC in OMNeT++

In this section, we will design the CRC implementation on data link layer. We will calculate CRC code for the source. This will be followed by the concatenation of all headers values and use that value with pre-calculated polynomial to calculate CRC code. The calculated CRC code will be appended with the message before transmission. At destination’s end, we will detach the CRC
code value with received message and check the code to verify whether the message is received correctly or not. To understand CRC in a Point-to-Point network, simulation steps are described below:

1- To create CRC project, follow steps (1—9) of section 4.2.
2- To create network description files (.ned), follow steps (2—14) of section 5.4.
3- To create TCP/IP packet, follow steps (15—17) of section 5.4. Write code shown in Listing 6.1 in packet.msg and to create respective implementation files (.cc), follow step 19 of section 5.4.

### Listing 6.1

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>packet TcpIpPacket {</code></td>
</tr>
<tr>
<td>2</td>
<td><code>int appHeader = 0;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>int tcpHeader = 0;</code></td>
</tr>
<tr>
<td>4</td>
<td><code>int ipHeader = 0;</code></td>
</tr>
<tr>
<td>5</td>
<td><code>int dllHeader = 0;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>int phyHeader = 0;</code></td>
</tr>
<tr>
<td>7</td>
<td><code>int crc = 0;</code></td>
</tr>
<tr>
<td>8</td>
<td><code>int data = 0;</code></td>
</tr>
<tr>
<td>9</td>
<td><code>string packetFormat = NULL;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Integer header fields are used to assign port numbers and addresses (i.e., tcp port number, IP and MAC addresses). CRC mechanism is applied on the complete packet (header + payload), as described in Listing 6.5. Here, the selection of integer type headers makes the computational process simple at each layer. For real-life network designs, any type of packet headers is converted to bit form for the computational process.

4- To create C++ source file (relevant to PhysicalLayer.ned, DataLinkLayer.ned, NetworkLayer.ned, TransportLayer.ned, ApplicationLayer.ned), follow steps 20—22 of section 5.4.

5- Write C++ code for ApplicationLayer as shown in Listing 6.2.

### Listing 6.2

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>#include &lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>#include &lt;string.h&gt;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>#include &lt;omnetpp.h&gt;</code></td>
</tr>
<tr>
<td>4</td>
<td><code>#include &quot;packet_m.h&quot;</code></td>
</tr>
<tr>
<td>5</td>
<td><code>using namespace omnetpp;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>class ApplicationLayer : public cSimpleModule{</code></td>
</tr>
<tr>
<td>7</td>
<td><code>protected:</code></td>
</tr>
<tr>
<td>8</td>
<td><code>virtual void initialize() override;</code></td>
</tr>
<tr>
<td>9</td>
<td><code>virtual void handleMessage(cMessage *msg) override;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>virtual TcpIpPacket* generateNextMessage();</code></td>
</tr>
<tr>
<td>11</td>
<td><code>};</code></td>
</tr>
<tr>
<td>12</td>
<td><code>Define_Module(ApplicationLayer);</code></td>
</tr>
<tr>
<td>13</td>
<td><code>void ApplicationLayer::initialize() {</code></td>
</tr>
<tr>
<td>14</td>
<td><code>// To initialize Message Passing from PC1</code></td>
</tr>
<tr>
<td>15</td>
<td><code>if(strcmp(par(&quot;AppID&quot;), operator)constchar *(), &quot;PC1&quot;) == 0){</code></td>
</tr>
<tr>
<td>16</td>
<td><code>scheduleAt (0.0, generateNextMessage());</code></td>
</tr>
</tbody>
</table>
At initialization phase of (lines 13—17 of Listing 6.2), network restricts the initialization of packet transmission from application layer having AppID = PC1 which is defined in omnetpp.ini (which will be explained in Listing 6.7) using comparison operator const char*(). Upon receiving a packet, Application layer checks the type of the packet. If the packet arrives at the input port (line 20), application layer just deletes that message. Here, deleted message is considered as successful message reception. However, if the message is a self-message (line 22), then it makes a new packet and sends it to the transport layer (line 25). For the generation of new application packet after a specific time interval, scheduleAt() function (line 15 and line 26) is used which have two arguments, time for packet scheduling and the packet which needs to be transmitted. In line 26 of Listing 6.2, simTime() + 0.1 provides scheduling of new message after 100ms of current simulation time. In line 30 of Listing 6.2, a function is defined to create a new application packet and sets appHdr and data information which is generated using built-in function uniform(int, int). The uniform(int, int) function creates random data between provided range (lines 32—33 of Listing 6.2). The generated message (line 39 of Listing 6.2) is scheduled (lines 15 and 26 of Listing 6.2) for further processing.
28- Write C++ code for TransportLayer as shown in Listing 6.3.

<table>
<thead>
<tr>
<th>Listing 6.3</th>
<th>Implementation of TransportLayer module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td>Line 2:</td>
<td>#include&lt;string.h&gt;</td>
</tr>
<tr>
<td>Line 3:</td>
<td>#include&lt;omnetpp.h&gt;</td>
</tr>
<tr>
<td>Line 4:</td>
<td>#include&lt;packet_m_h&gt;</td>
</tr>
<tr>
<td>Line 5:</td>
<td>#include&lt;vector&gt;</td>
</tr>
<tr>
<td>Line 6:</td>
<td>using namespace omnetpp;</td>
</tr>
<tr>
<td>Line 7:</td>
<td>class TransportLayer : public cSimpleModule{</td>
</tr>
<tr>
<td>Line 8:</td>
<td>protected:</td>
</tr>
<tr>
<td>Line 9:</td>
<td>virtual void initialize() override;</td>
</tr>
<tr>
<td>Line 10:</td>
<td>virtual void handleMessage(cMessage *msg) override;</td>
</tr>
<tr>
<td>Line 11:</td>
<td>};</td>
</tr>
<tr>
<td>Line 12:</td>
<td>Define_Module(TransportLayer);</td>
</tr>
<tr>
<td>Line 13:</td>
<td>void TransportLayer::initialize(){</td>
</tr>
<tr>
<td>Line 14:</td>
<td>void TransportLayer::handleMessage(cMessage *msg){</td>
</tr>
<tr>
<td>Line 15:</td>
<td>TcpIpPacket *pkt = check_and_cast&lt;TcpIpPacket *&gt;(msg);</td>
</tr>
<tr>
<td>Line 16:</td>
<td>if(pkt-&gt;arrivedOn(&quot;tcpGate$i&quot;,0)){</td>
</tr>
<tr>
<td>Line 17:</td>
<td>char packetInfo [40];</td>
</tr>
<tr>
<td>Line 18:</td>
<td>pkt-&gt;setTcpHeader(uniform(1,6));</td>
</tr>
<tr>
<td>Line 19:</td>
<td>sprintf(packetInfo,&quot;TCP Header [%d] added&quot;,</td>
</tr>
<tr>
<td>Line 20:</td>
<td>pkt-&gt;getTcpHeader());</td>
</tr>
<tr>
<td>Line 21:</td>
<td>pkt-&gt;setPacketFormat(packetInfo);</td>
</tr>
<tr>
<td>Line 22:</td>
<td>*pkt = tcpIpPacket(packetInfo,0);</td>
</tr>
<tr>
<td>Line 23:</td>
<td>bubble(packetInfo);</td>
</tr>
<tr>
<td>Line 24:</td>
<td>send(pkt, &quot;tcpGate$o&quot;,1);</td>
</tr>
<tr>
<td>Line 25:</td>
<td>if(pkt-&gt;arrivedOn(&quot;tcpGate$i&quot;,1)){</td>
</tr>
<tr>
<td>Line 26:</td>
<td>pkt-&gt;setTcpHeader(0);</td>
</tr>
<tr>
<td>Line 27:</td>
<td>send(pkt, &quot;tcpGate$o&quot;,0);</td>
</tr>
<tr>
<td>Line 28:</td>
<td>}</td>
</tr>
<tr>
<td>Line 29:</td>
<td>}</td>
</tr>
</tbody>
</table>

At handling phase (lines 14—29 of Listing 6.3), Transport layer checks whether it received a packet from the Application layer or the Network layer. Upon receiving a packet from Network layer (line 25), it removes TCP header and sends it to the Application layer (line 26). In case, Transport layer receives the packet (line 16) from the Application layer, it adds tcpheader and forwards the packet to the Network layer (line 23).

29- Write C++ code for NetworkLayer as shown in Listing 6.4.

<table>
<thead>
<tr>
<th>Listing 6.4</th>
<th>Implementation of NetworkLayer module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td>Line 2:</td>
<td>#include&lt;string.h&gt;</td>
</tr>
<tr>
<td>Line 3:</td>
<td>#include&lt;omnetpp.h&gt;</td>
</tr>
<tr>
<td>Line 4:</td>
<td>#include&lt;packet_m_h&gt;</td>
</tr>
</tbody>
</table>
At handling phase (lines 14—27 of Listing 6.4), Network layer checks whether it received a packet from the Data Link layer or from the Transport layer. Upon receiving a packet from the Data Link layer (line 23), it removes IP header and sends it to the Transport layer (line 25) and if it received a packet from the Transport layer (line 16), it adds IP header and sends it to Data Link layer (line 22).

30- Write C++ code for the DataLinkLayer as shown in Listing 6.5.

<table>
<thead>
<tr>
<th>Listing 6.5</th>
<th>Implementation of DataLinkLayer module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td><code>#include&lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td>Line 2:</td>
<td><code>#include&lt;string.h&gt;</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>#include&lt;omnetpp.h&gt;</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>#include&lt;packet_m_h&gt;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>using namespace omnetpp;</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>class DataLinkLayer : public cSimpleModule{</code></td>
</tr>
<tr>
<td>Line 7:</td>
<td><code>private:</code></td>
</tr>
<tr>
<td>Line 8:</td>
<td><code>int headersConcatenation;</code></td>
</tr>
<tr>
<td>Line 9:</td>
<td><code>const int crcDivider =13;</code></td>
</tr>
<tr>
<td>Line 10:</td>
<td><code>int crcValue;</code></td>
</tr>
<tr>
<td>Line 11:</td>
<td><code>protected:</code></td>
</tr>
</tbody>
</table>
virtual void initialize() override;
virtual void handleMessage(cMessage *msg) override;

Define_Module(DataLinkLayer);

void DataLinkLayer::initialize()
{
}

DataLinkLayer::handleMessage(cMessage *msg)
{
    TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
    if(pkt->arrivedOn("dllGate$i", 0))
    {
        pkt->setDllHeader(uniform(1, 6));
        headersConcatenation = (pkt->getAppHeader() +
                                pkt->getData() * 10 +
                                pkt->getTcpHeader() * 100 +
                                pkt->getIpHeader() * 1000 +
                                pkt->getDllHeader() * 10000);

        crcValue = crcDivider - ((headersConcatenation * 100) % crcDivider);
        pkt->setCRC(crcValue);
        bubble("CRC Added");
        send(pkt, "dllGate$o", 1);
    }
    else
    {
        bubble("CRC checked NO error");
        send(pkt, "dllGate$o", 0);
    }
}

// For packet receiving from Network Layer
if(pkt->arrivedOn("dllGate$i", 1))
{
    // Setting of Data Link layer header
    pkt->setDllHeader(uniform(1, 6));
    /* Concatenation of all header values + data for CRC code Calculation */
    headersConcatenation = (pkt->getAppHeader() +
                            pkt->getData() * 10 +
                            pkt->getTcpHeader() * 100 +
                            pkt->getIpHeader() * 1000 +
                            pkt->getDllHeader() * 10000);

    // Calculation of CRC code
    crcValue = crcDivider - ((headersConcatenation * 100) % crcDivider);
    // Attach CRC code in packet before transmission
    pkt->setCRC(crcValue);
    // Send packet to Physical Layer
    bubble("CRC Added");
    send(pkt, "dllGate$o", 1);
}

// For packet receiving from Physical Layer
if (pkt->arrivedOn("dllGate$i", 1))
{
    int error = uniform(1, 4);
    if(error!= 2)
    {
        pkt->setAppHeader(7);
        // At receiver side, concatenation of all header values + data for CRC */
        headersConcatenation = (pkt->getAppHeader() +
                                pkt->getData() * 10 +
                                pkt->getTcpHeader() * 100 +
                                pkt->getIpHeader() * 1000 +
                                pkt->getDllHeader() * 10000);

        // Applying CRC check on received frame
        if((headersConcatenation * 100 +
            pkt->getCRC()) % crcDivider == 0){
            // No Error Message Display
            bubble("CRC checked NO error");
            send(pkt, "dllGate$o", 0);
        }
        else{
            bubble("ERROR Detected");
            delete pkt;
        }
    }
}
At handling phase (lines 14—27 of Listing 6.5), the Data Link layer checks whether it received the packet from a Physical layer or Network layer (line 19). Upon receiving from Network layer, it adds DLL Header, adds CRC code and sends it to the Network layer. CRC divider (line 9 of Listing 6.5) is used for CRC code generation. On ethernet cards, CRC divider value is generated using a standard polynomial. Source node creates the CRC code (line 22 of Listing 6.5) and attaches it to received packet from Network layer. Evaluation of CRC value (line 23 of Listing 6.5) is done by concatenating all headers values coming from Network layer in respective order and make a single value to formulate it with CRC divider value. Concatenation process mimics the scenario of binary bit stream used for CRC code evaluation as discussed in section 6.2.

On the other hand, if the packet is received from the Physical layer (line 27 of Listing 6.5), it performs a Checksum on the data packet. Checksum is done by taking received CRC code value and concatenates all headers as done on transmit side. The receiver performs a check whether the received packet values are same as transmitted (line 32 of Listing 6.5). In case, CRC evaluation result is zero, then Data Link layer ensures error-free packet reception. For code simplicity, an artificial random packet error has been introduced by changing appID of the source node. This introduced error is checked at Data Link layer (lines 28—30 of Listing 6.5) of the destination node. If the packet is without error then it is forwarded to the Network layer (line 34 of Listing 6.5). Here, packet re-send mechanism is not discussed in the case of error packet reception. However, packet re-send mechanism is addressed in Chapter 7.

31- Write C++ code for PhysicalLayer as shown in Listing 6.6.

**Listing 6.6 Implementation of PhysicalLayer module**

```cpp
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include <packet_m.h>
using namespace omnetpp;

class PhysicalLayer : public cSimpleModule{
    protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(PhysicalLayer);

void PhysicalLayer::initialize()
{
    
    void PhysicalLayer::handleMessage(cMessage *msg)
    {
        TcpIpPacket *pkt = check_and_cast<TcpIpPacket*>(msg);
        // For packet receiving from Data Link Layer
        if(pkt->arrivedOn("phyGate$i",0))
        {
            char packetInfo [40];
            // For setting and sending of IP Packet
            pkt->setPhyHeader (uniform(1,6));
            sprintf(packetInfo,"PHY Header [%d] Added ",
                    pkt->getPhyHeader());
            pkt->setPacketFormat(packetInfo);
            bubble(pkt->getPacketFormat());
            // Send packet on PC port
            send(pkt, "phyGate$o",1);
        }
        // Packet reception on PC port
        else if(pkt->arrivedOn("phyGate$i",1))
        {
```
After handling phase (lines 14—28 of Listing 6.6), Physical layer checks whether it received the packet from Data Link layer or Physical layer of connected PC. Upon receiving the packet from the Physical layer of connected PC (line 24), it removes Physical layer header and forwards the packet to Data Link layer (line 26). However, if the physical layer received the packet from Data Link layer (line 15), Physical layer adds header and forwards the packet to the connected PC (line 22).

32- To create a network initialization file (omnetpp.ini), right click on the OMNET++_CRC_Working folder and then select New → Initialization File (.ini) as shown in Figure 6.1.

33- Select OMNET++_CRC_Working and write file name as omnetpp.ini (if not already written) in File name text field and proceed with Next button (Figure 6.2 (a)).

34- Empty Ini File option appears in Select template text area and press Next (Figure 6.2 (b)).

35- Press browse button to select network in New Ini File window (Figure 6.2 (c)).

36- From available list, select NED type related to your project as shown in (Figure 6.2 (d)).

37- The selected NED type will appear in the text field (Figure 6.2 (e)).

38- Click finish to complete the creation of omnetpp.ini (Figure 6.2 (f)) and can be viewed in omnetpp.ini text editor (Figure 6.2 (g)).
To assign MAC addresses to module and ID to each compound module, type following lines in omnetpp.ini file.

Listing 6.7: Attributes of omnetpp.ini

Line 1: [General]
Line 2: network= omnet_crc_working.NEDSrc.Pt2Pt_Network
Line 3: **.PC1.appL.AppID = "PC1"
Line 4: **.PC2.appL.AppID = "PC2"

Whenever a simulation starts in OMNeT++, it reads NED and configuration file (omnetpp.ini) to assign parameter values, respectively. NED files are first preference, however, if NED files contain no fixed values for parameters, then configuration file option is used to assign. Here, in Listing 6.7, Application IDs are provided in the configuration file.

The project files can be seen in Project Explorer (Figure 6.3).
Right click on the OMNET++_CRC_Working folder icon in Project Explorer window and Build Project.

![Project Explorer](image)

**Figure 6.3:** Project files in project explorer window

The building of project may take several minutes and then the project will be ready to run. Run the project using play button in the menu bar. Simulation Tekenv window will appear and then Run the simulation using run button in the menu bar.

We have implemented CRC code generation for error detection at receiving end. The simulation shows that CRC code is generated at data link layer based on the packet received from network layer and both are forwarded to other PC. At receiving end CRC code is evaluated at data link layer, in the case of match packet forwarded to upper layers otherwise data link layer reports the error message.

**Practice Task**

1. Create a network having three nodes as shown in Figure 6.4. Both PC1 and PC2 nodes are required to send Data packets (including CRC block) to each other via intermediate node R. On reception of a packet, intermediate node R performs CRC check at MAC layer and discards all those packets whose CRC indicates any transmission errors.

![Network topology](image)

**Figure 6.4:** Network topology for practice task 1
Implementation of ARQ Stop-and-Wait

Learning Objectives:

After studying this chapter, you will be able to describe:

- the concept of reliable transmission
- the concept of Automatic Repeat reQuest (ARQ)
- the concept of ARQ Stop-and-Wait approach
- Implementation of ARQ Stop-and-Wait approach
7.1 Reliable Transmission – An Overview

Depending on the application nature, various mechanisms for reliable transmission can be deployed at Data Link layer, Transport layer and in some cases at the Application layer. Irrespective of deployment layer, it is important to understand the fundamental objectives associated with reliable transmission schemes. In addition to CRC, an effective reliable transmission scheme must achieve following objectives:

- Channel reliability
- Utilizing full channel capacity
- In-order delivery of packets

Automatic Repeat request (ARQ) is an approach for reliable transmission, which is actually based on the notion of sending acknowledgments whenever a frame is received. In other words, any time an error is detected during an exchange, specified frames are retransmitted. Stop-and-Wait and Sliding Window algorithms are two variants of ARQ mechanism.

In this chapter, we have explained the implementation of ARQ stop-and-wait mechanism in OMNeT++.

7.2 Stop-and-Wait Approach

Stop-and-Wait is considered one of the simplest ARQ-based schemes. In this scheme, the sender sends a single frame to the receiver and then waits for an acknowledgment (ACK). The sender sends the next frame if and only if it has already received the ACK of the previous frame. The sender keeps the copy of each frame till its ACK is received. If the sender is not able to receive the ACK after a certain period of time, the sender timeout event is triggered and generates an ARQ while transmitting the frame again. Data frames and acknowledgment frames are numbered differently for the purpose of identification otherwise sender may lose track of the frames it has already received acknowledgment. As shown in Figure 7.1, a sender continues sending one frame at a time if it continues to receive ACK from the receiver. The sender starts a timer whenever it transmits a frame. If the receiver is unable to receive the frame sent by the sender, then the sender will receive no ACK. This will trigger a timeout event at the sender after a particular duration has elapsed and the sender will retransmit the original frame. On the other hand, if a frame sent by sender does reach the destination, and an error occurs during the transmission of the ACK, the sender will react in a similar manner and retransmits the previous frame after the time out. The sender continues this process of sending a frame and waiting for an acknowledgment as long as the sender has data to send. When the sender decides to finish the transmission, it sends a special frame (End-of-Transmission frame) to indicate the termination of the transmission.

![Figure 7.1: ARQ – Stop-and-Wait mechanism](image)

The main advantage of the stop-and-wait scheme lies in the simplicity of the scheme and ultimately meets two objectives of reliable transmission i.e., the appearance of channel reliability.
and in-order delivery of packets. However, stop-and-wait is not suitable because of inefficient utilization of channel capacity and suffers from long transmission delays. If the two communicating devices are a distance apart, then a considerable amount of time is wasted for a single frame to be received at the destination and then acknowledgment to be received by the sender.

7.3 Implementation of ARQ Stop-and-Wait

In this chapter, we will design an ARQ stop and wait mechanism in which packets are sent to the destination with unique ID (sequence number). Destination will reply the ACK of received packet with the same ID to the source node. After successful reception of corresponding, ACK source node will send next message with new ID. In the case of packet or ACK reception, failure source node will resend the same packet after a certain period of time. To understand ARQ Stop-and-Wait through simulation in a Point-to-Point network, follow steps as given below:

1- To create OMNeT++ project, follow steps (1—9) of section 4.2.
2- To create network description files (.ned), follow steps (2—14) of section 5.4.
3- To create TCP/IP packet, follow steps (15—17) of section 5.4. Write code shown in Listing 7.1 in `packet.msg` and to create respective implementation files (.cc), follow step 19 of section 5.4.
4- Write packet attributes and save as `packet.msg` as shown in Listing 7.1.

**Listing 7.1** Attributes of TcpIpPacket

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>packet</code> TcpIpPacket{</td>
</tr>
<tr>
<td>2</td>
<td><code>int</code> srcAdd = 0;</td>
</tr>
<tr>
<td>3</td>
<td><code>int</code> destAdd = 0;</td>
</tr>
<tr>
<td>4</td>
<td><code>string</code> receivedMessage = NULL;</td>
</tr>
<tr>
<td>5</td>
<td><code>string</code> sendMessage = NULL;</td>
</tr>
<tr>
<td>6</td>
<td><code>string</code> appHeader = NULL;</td>
</tr>
<tr>
<td>7</td>
<td><code>string</code> tcpHeader = NULL;</td>
</tr>
<tr>
<td>8</td>
<td><code>string</code> ipHeader = NULL;</td>
</tr>
<tr>
<td>9</td>
<td><code>string</code> dillHeader = NULL;</td>
</tr>
<tr>
<td>10</td>
<td><code>string</code> phyHeader = NULL;</td>
</tr>
<tr>
<td>11</td>
<td><code>string</code> receivedMessageAck = NULL;</td>
</tr>
<tr>
<td>12</td>
<td><code>string</code> sendMessageAck = NULL;</td>
</tr>
<tr>
<td>13</td>
<td><code>bool</code> ack = false;</td>
</tr>
<tr>
<td>14</td>
<td>}</td>
</tr>
</tbody>
</table>

5- To create C++ source file (relevant to `PhysicalLayer.ned`, `DataLinkLayer.ned`, `NetworkLayer.ned`, `TransportLayer.ned`, `ApplicationLayer.ned`), follow steps 20—22 of section 5.4.
6- Write C++ code for ApplicationLayer as shown in Listing 7.2.

**Listing 7.2** Implementation of ApplicationLayer module

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>#include&lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>#include&lt;string.h&gt;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>#include&lt;omnetpp.h&gt;</code></td>
</tr>
<tr>
<td>4</td>
<td><code>#include&quot;packet_m.h&quot;</code></td>
</tr>
<tr>
<td>5</td>
<td><code>using namespace omnetpp;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>class ApplicationLayer : public cSimpleModule{</code></td>
</tr>
<tr>
<td>7</td>
<td><code>private:</code></td>
</tr>
<tr>
<td>8</td>
<td><code>// variable used to assign unique ID for each packet</code></td>
</tr>
<tr>
<td>9</td>
<td><code>int packetSequenceNo=0;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>TcpIpPacket* pkt;</code></td>
</tr>
<tr>
<td>11</td>
<td><code>TcpIpPacket* resendPacket;</code></td>
</tr>
</tbody>
</table>
protected:
virtual void initialize() override;
virtual void handleMessage(cMessage *msg) override;
virtual TcpIpPacket *generateNextMessage();
};
Define_Module(ApplicationLayer);

void ApplicationLayer::initialize()
    { //To initialize Message Passing from PC1
        if(strcmp(par("AppID")) \operator<constchar *(),"PC1")==0){
            //Self message to be generated by Module "PC1"
            scheduleAt (0.0, generateNextMessage());
        }
    }

//Implementation of overriding handleMessage() function
void ApplicationLayer::handleMessage(cMessage *msg){
    pkt = check_and_cast<pkt *>(msg);
    //For initial transmission or retransmission
    if(!pkt->getAck()){
        //On message arrival at input port of Application Layer
        if(pkt->arrivedOn("appGate$i",0)){
            //To cancel self-message timer. No need to retransmit.
            cancelEvent(resendPacket);
            delete pkt;
        }
    }
    pkt *packetToSend =generateNextMessage();
    //Making of duplicate packets for retransmission (if requires)
    resendPacket = packetToSend->dup();
    //To send packet to Transport layer
    send(packetToSend,"appGate$o",0);
    /* Upon not receiving ACK within 1 second, resend duplicate packet (line 30) */
    scheduleAt (simTime()+1.0, resendPacket);
    // Self-message (To initialize or to resend packet scenario)
    //Making of duplicate packet
    }else{
        resendPacket= pkt->dup();
        /* Upon not receiving ACK within 1 second, resend duplicate packet */
        scheduleAt(simTime()+1.0, resendPacket);
        //To send message to Transport layer
        bubble(pkt->getSendMessage());
        send(pkt,"appGate$o",0);
    }
    //Upon message reception at Application layer
    }else{
        bubble(pkt->getReceivedMessage());
    }
}

TcpIpPacket* ApplicationLayer::generateNextMessage(){
    //To check if Message Sequence No. is 0 (sequenceNo is 0 or 1)
    if(packetSequenceNo== 0){
At initializing phase (line 18—21 Listing 7.2), network restricts to initiate packet transmission from Application layer having AppID = PC1 which is defined in omnetpp.ini (Listing 7.7). Application layer generates new messages. The generateMessage() function is responsible to attach sequence number (0 or 1) (line 52 and 63). At handling phase (line 23—43 Listing 7.2), Application layer checks whether it receives a packet from Transport layer or it is a self-message. Self-message initiated by the Application layer depicts new messages (line 33) for Transport layer (line 33) and the same message is rescheduled in the case of ACK failure (line 35). However, in re-transmission scenario (line 32 and 35), packet resend at a time out. Re-scheduled packets are canceled (line 26) in the case of successful ACK reception (line 24). Upon message received from Transport Layer (line 25 Listing 7.2), packet is considered as received and reschedule event (line 32 Listing 7.2) is cancelled (line 26 Listing 7.2) and a new message is generated for transmission (line 28 Listing 7.2) and again repeat the rescheduling process before sending to the Transport layer.

7- Write C++ code for TransportLayer.cc file as shown in Listing 7.3.

### Listing 7.3 Implementation of TransportLayer module

```cpp
#include<stdio.h>
#include<string.h>
#include<omnetpp.h>
#include<packet_m.h>
#include<vector>
using namespace omnetpp;

class TransportLayer : public cSimpleModule{
protected:
    virtual void initialize() override;
```
Line 10:   virtual void handleMessage(cMessage *msg) override;
Line 11:  }

Line 12: Define_Module(TransportLayer);

Line 13: void TransportLayer::initialize()
Line 14: {

Line 15: void TransportLayer::handleMessage(cMessage *msg){
Line 16:   TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
Line 17:   if(pkt->arrivedOn("tcpGate$i",0)){
Line 18:     send(pkt, "tcpGate$o",1);
Line 19:   };

Line 20:   }

Line 21:   }

Line 22: }

At handling phase (line 14—22 of Listing 7.3), Transport layer checks whether it receives a packet from Application layer or Network layer. Upon receiving a packet from Network layer (line 19), it removes tcpHeader and sends it to the Application layer (line 20) and if it receives a packet from the Application layer (line 15), it adds tcpHeader and sends it to the Network layer.

8- Write following C++ code for NetworkLayer.cc file.

```
Listing 7.4 Implementation of NetworkLayer module

Line 1: #include<stdio.h>
Line 2: #include<string.h>
Line 3: #include<omnetpp.h>
Line 4: #include<packet_m.h>
Line 5: #include<vector>
Line 6: using namespace omnetpp;

Line 7: class NetworkLayer : public cSimpleModule
Line 8: {
Line 9:   protected:
Line 10:     virtual void initialize() override ;
Line 11:     virtual void handleMessage(cMessage *msg) override;
Line 12:   };

Line 13: Define_Module(NetworkLayer);

Line 14: void NetworkLayer:::handleMessage(cMessage *msg){
Line 15:   TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
Line 16:   if(pkt->arrivedOn("ipGate$i",0)){
Line 17:     send(pkt, "ipGate$o",1);
Line 18:   };
```
if(pkt->arrivedOn("ipGate$i",1)){
    //To remove IP header of the PC in this layer(IP Address)
    pkt->setIpHeader(NULL);
    // To send packet to Transport layer
    send(pkt, "ipGate$o",0);
}
}

At handling phase (line 14–23 of Listing 7.4), Network layer checks whether it receives a packet from Data Link layer or Transport layer. Upon receiving a packet from Data Link layer (line 19), it removes IP header and sends it to the Transport layer (line 21) and if receives a packet from Transport layer (line 16), it adds IP header and sends it to Data Link layer (line 17).

9- Write C++ code for DataLinkLayer.cc file as shown in Listing 7.5.

<table>
<thead>
<tr>
<th>Listing 7.5</th>
<th>Implementation of DataLinkLayer module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td>Line 2:</td>
<td>#include&lt;string.h&gt;</td>
</tr>
<tr>
<td>Line 3:</td>
<td>#include&lt;omnetpp.h&gt;</td>
</tr>
<tr>
<td>Line 4:</td>
<td>#include&lt;packet_m.h&gt;</td>
</tr>
<tr>
<td>Line 5:</td>
<td>using namespace omnetpp;</td>
</tr>
<tr>
<td>Line 6:</td>
<td>class DataLinkLayer : public cSimpleModule{</td>
</tr>
<tr>
<td>Line 7:</td>
<td>cPar* macAddress;</td>
</tr>
<tr>
<td>Line 8:</td>
<td>protected:</td>
</tr>
<tr>
<td>Line 9:</td>
<td>virtual void initialize() override ;</td>
</tr>
<tr>
<td>Line 10:</td>
<td>Virtual void handleMessage(cMessage *msg) override;</td>
</tr>
<tr>
<td>Line 11:</td>
<td>};</td>
</tr>
<tr>
<td>Line 12:</td>
<td>Define_Module(DataLinkLayer);</td>
</tr>
<tr>
<td>Line 13:</td>
<td>void DataLinkLayer::initialize(){</td>
</tr>
<tr>
<td>Line 14:</td>
<td>void DataLinkLayer::handleMessage(cMessage *msg){</td>
</tr>
<tr>
<td>Line 15:</td>
<td>TcpIpPacket *pkt = check_and_cast&lt;TcpIpPacket *&gt;(msg);</td>
</tr>
<tr>
<td>Line 16:</td>
<td>macAddress = 5par(&quot;macAddress&quot;);</td>
</tr>
<tr>
<td>Line 17:</td>
<td>if(pkt-&gt;arrivedOn(&quot;dllGate$i&quot;,0)){</td>
</tr>
</tbody>
</table>
| Line 18:    |             //To make sure ACK flag is false
| Line 19:    |             pkt->setAck(false);         |
| Line 20:    |             //This variable is used to create Random Number from 1 to 3
| Line 21:    |             int randomNumber = uniform(1,3); |
| Line 22:    |             //Valid MAC will assign only when Random Number value is 2
| Line 23:    |             if(randomNumber == 2)       |
| Line 24:    |                 //Setting of destination MAC address
| Line 25:    |                 pkt->setDllHeader("AC-F4-BB-7B-57-51"); |
| Line 26:    |             //In this case, ARQ implementation will be Activated
| Line 27:    |             else                      |
| Line 28:    |                 pkt->setDllHeader("NULL"); |
| Line 29:    |             //Send packet to Physical layer
| Line 30:    |             send(pkt, "dllGate$o",1);   |
| Line 31:    |     }                                 |
| Line 32:    |     else if(pkt->arrivedOn("dllGate$i",1)){ |
| Line 33:    |         //On packet arrival from Physical layer
| Line 34:    |     }                                 |
| Line 35:    | }                                     |
// Confirm Network layer of destination node
Line 27:    if(!pkt->getAck()){
            // Match own Data Link layer header with incorporated MAC address
Line 28:        if(strcmp(pkt->getDllHeader(),
                           macAddress->stringValue()) == 0){
                // In case of MAC matched, create ACK for received data packet
Line 29:            TcpIpPacket *messagePacketAck = new
Line 30:                TcpIpPacket(pkt->getSendMessageAck(),4);
Line 31:            messagePacketAck->setReceivedMessageAck(pkt->
Line 32:                getReceivedMessageAck());
Line 33:            // To ensure upper layers successful packet retrieval
Line 34:               pkt->setAck(true);
Line 35:        }else{
Line 36:            delete pkt;
Line 37:            bubble("Packet Dropped");
Line 38:        }
Line 39:    }

    // To receive ACK on source node
Line 40:    else if(pkt->getAck()){
            // To ensure upper layer that it’s successfully received ACK
Line 41:        pkt->setAck(false);
            // Send ACK to Network layer
Line 42:        send(pkt,"dllGate$o",0);
Line 43:        bubble(pkt->getReceivedMessageAck());
Line 44:    }
Line 45:}
Line 46:}

At handling phase (line 14—46 of Listing 7.5), Data Link layer checks whether it receives a packet from Network layer or Physical layer. Upon receiving a packet from Network Layer (line 17 of Listing 7.5) Data Link layer header of the destination node is set (line 21 of Listing 7.5) and sent to Physical layer (line 24 of Listing 7.5). Here, we have generated an error by changing MAC address randomly (not for each packet) by using uniform(int, int) function (line 19 of Listing 7.5). If function generates random value 2, then condition at line 20 will be valid and correct MAC address will be assigned. Otherwise, MAC address will be assigned NULL (line 23). This error will be caught on destination Data Link layer (line 27) by ARQ mechanism. At the destination node, the packet will be received by the physical layer (line 26 of Listing 7.5), which may be a data packet or acknowledgment packet. Packet type is checked at (line 27 and 40 of Listing 7.5) for data and acknowledgment respectively. For destination node, the packet must be a data packet (line 27 of Listing 7.5). MAC address of destination node (line 16 of Listing 7.5) is compared with the MAC address assigned to the source node using strcmp function (line 28 of Listing 7.5). Upon MAC address matching two processes are done. First, an acknowledgment sent to the source node through the Physical layer (line 34) and second is to send received a message to the Network layer (line 33 of Listing 7.5). If MAC addresses are not matched (line 35 of Listing 7.5) it will be considered as an error in data packet reception. Therefore, destination node will discard the received packet (line 36 of Listing 7.5). When receiver node will get acknowledgment packet from the Physical layer (line 40), then ACK flag will be set false and sent to the network layer (line 42 of Listing 7.5).
Write C++ code for PhysicalLayer.cc file as shown in Listing 7.6.

**Listing 7.6** Implementation of PhysicalLayer module

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td>2</td>
<td>#include&lt;string.h&gt;</td>
</tr>
<tr>
<td>3</td>
<td>#include&lt;omnetpp.h&gt;</td>
</tr>
<tr>
<td>4</td>
<td>#include&lt;packet_m.h&gt;</td>
</tr>
<tr>
<td>5</td>
<td>using namespace omnetpp;</td>
</tr>
<tr>
<td>6</td>
<td>class PhysicalLayer : public cSimpleModule{</td>
</tr>
<tr>
<td>7</td>
<td>protected:</td>
</tr>
<tr>
<td>8</td>
<td>virtual void initialize() override;</td>
</tr>
<tr>
<td>9</td>
<td>virtual void handleMessage(cMessage *msg) override;</td>
</tr>
<tr>
<td>10</td>
<td>};</td>
</tr>
<tr>
<td>11</td>
<td>Define_Module(PhysicalLayer);</td>
</tr>
<tr>
<td>12</td>
<td>void PhysicalLayer::initialize(){ }</td>
</tr>
<tr>
<td>13</td>
<td>void PhysicalLayer::handleMessage(cMessage *msg){</td>
</tr>
<tr>
<td>14</td>
<td>TcpIpPacket <em>pkt = check_and_cast&lt;TcpIpPacket</em>&gt;(msg);</td>
</tr>
<tr>
<td>15</td>
<td>// On packet arrival from Data Link layer</td>
</tr>
<tr>
<td>16</td>
<td>if(pkt-&gt;arrivedOn(&quot;phyGate$i&quot;,0)){</td>
</tr>
<tr>
<td>17</td>
<td>// To send packet to the channel</td>
</tr>
<tr>
<td>18</td>
<td>send(pkt, &quot;phyGate$o&quot;,1);</td>
</tr>
<tr>
<td>19</td>
<td>}</td>
</tr>
<tr>
<td>20</td>
<td>// On packet arrival from other PC</td>
</tr>
<tr>
<td>21</td>
<td>else if(pkt-&gt;arrivedOn(&quot;phyGate$i&quot;,1)){</td>
</tr>
<tr>
<td>22</td>
<td>//To remove Physical Layer header</td>
</tr>
<tr>
<td>23</td>
<td>pkt-&gt;setPhyHeader(NULL);</td>
</tr>
<tr>
<td>24</td>
<td>//Send packet to Data Link layer DLL</td>
</tr>
<tr>
<td>25</td>
<td>send(pkt, &quot;phyGate$o&quot;,0);</td>
</tr>
</tbody>
</table>

At handling phase (line 14-22 of Listing 7.6), Physical layer checks whether it receives a packet from Data Link layer or Physical layer of connected PC. Upon receiving a packet from the Physical layer (line 18) of connected PC, it removes Phyheader and sends it to Data Link layer (line 20) and if receives a packet from Data Link layer (line 15), it adds Phyheader and sends it to connected PC (line 16).

To create a network initialization file (**omnetpp.ini**), right click on the **TCP/IPStack** folder and then select **New → Initialization File (.ini)** as shown in Figure 7.2.

Select **TCP/IPStack** and write file name as **omnetpp.ini** (if not already written) in **File name** text field and proceed with **Next** button (Figure 7.3 (a)).

Select **Empty Ini File** shown in **Select template** text area and press **Next** (Figure 7.3 (b)). Press browse button to select network in **New ini File** window (Figure 7.3 (c)). From available list, select NED type related to your project as shown in Figure 7.3 (d). The selected NED type will appear in the text field (Figure 7.3 (e)). Click **Finish** to complete the creation of **omnetpp.ini** (Figure 7.3 (f)). Text view of omnetpp.ini file is shown in Figure 7.3 (g).
Figure 7.2: Selection of initialization file (ini) from the main menu

(a) (b) (c) (d) (e) (f)
To assign MAC addresses to module and ID to each compound module, write `omnetpp.ini` file as shown in Listing 7.7.

**Listing 7.7**

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>[General]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2:</td>
<td><code>network = tcpipstack.NEDSrc.Pt2Pt_Network</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>**.PC1.d1L.macAddress = &quot;EA-E3-AB-3A-27-5F&quot;</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>**.PC2.d1L.macAddress = &quot;AC-F4-BB-7B-57-51&quot;</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>**.PC1.appL.AppID = &quot;PC1&quot;</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>**.PC2.appL.AppID = &quot;PC2&quot;</code></td>
</tr>
</tbody>
</table>

The project files can be seen in Project Explorer window (Figure 7.4).

This simulation shows the process of message sending from source node to the destination using ARQ mechanism. Data link layer at receiving node generated the random error scenario in which packet drop shows that there is a failure in packet reception. In this situation, course node waits for ACK for the specific time interval and resend the packet. In the case of successful packet reception destination node sends ACK to the source node through data link layer and forwards the packet to the upper layer. At source when ACK is received then a new message is forwarded to the destination node.
Practice Task

1. Create a network having three nodes as shown in Figure 7.5. Both PC1 and PC2 nodes are required to send Data packets (including CRC block) to each other via intermediate node R. A packet is dropped at the receiving node if its CRC check indicates any transmission errors. On reception of a data packet, the receiving node (PC1, PC2, or R node) should send back acknowledgment packet to the sender node. The sender node will wait for maximum 10ms for the acknowledgment of the sent packet. On time-out (after 10 ms), the sender node should re-send the same data packet.

![Figure 7.5: Network topology for practice task 1](image-url)
Implementation of Sliding Window

Learning Objectives:

After studying this chapter, you will be able to describe:

- the basics of Sliding Window Algorithm
- the working of Sliding Window Algorithm
- the implementation of Sliding Window Algorithm
8.1 Sliding Window Algorithm

As discussed in the previous chapter, Stop-and-Wait scheme suffers from long delays or channel underutilization problem. The reason behind this problem is that a sender has to wait for the acknowledgment for each packet separately. Sliding Window algorithm is proposed to overcome these shortcomings of the stop-and-wait scheme. Sliding Window algorithm suggests sending multiple frames by sender followed by an acknowledgment by sent by receiver [57]. Sliding Window algorithm is mostly used in connection-oriented network protocols including TCP/IP. Like Stop-and-Wait scheme, Sliding Window algorithm also works using DATA and ACK frames while assuming a full duplex channel. Data and ACK frames are verified using a CRC for successful transmission. Both sender and receiver keep track of a window for data frames and respective acknowledgments.

The Sliding Window algorithm identifies limits on the number of data frames sent before it starts waiting for an acknowledgment back from the receiver. The number of data frames sent consecutively before first ACK received is called the window size. The limits on window size may vary depending on the rate at which the receiver can process the data packets, and on the capacity of its buffer.

If the receiver is unable to process the data frames in the expected time assumed by the sender, the acknowledgment from the receiver will indicate the sender to decrease the number of frames in the window size for the next transmission. In extreme situations, the receiver may signal sender to suspend transmission to free up space in its buffer. Alternatively, if the receiver can process the data frame faster than the sender is sending, the acknowledgments will let know the sender to increase the number of packets in the next transmission window.

8.2 Working of Sliding Window Algorithm

To understand the working of Sliding Window algorithm, we have to consider the terminology associated with Sliding Window. In Sliding Window algorithm, the sender stamps each frame with an increasing sequence number starting from 0 to the maximum value that represents the window size. On the sender side, Send Window Size (SWS) represents the upper bound number of unacknowledged frames that a sender may transmit. LAR and LFS represent the sequence number of the Last ACK Received from the receiver and the Last Frame Sent by the sender, respectively.

Figure 8.1 shows 10 packets (having ID 21-30) in the sender window. ACK for 3 packets (with ID 21-23 shown in green) has been received by the sender and ACK for 5 packets (ID 24-28 shown in blue) is still awaited. Correspondingly, SWS should be less than equal to the difference of LFS and LAR (i.e., LFS – LAR \( \geq \) SWS) as shown in Figure 8.1.

On the receiver’s side, ACK for 3 packets (shown in green) has already been transmitted and the 3 packets (ID 24, 26, 27) have been received and their ACK is being sent. However, 2 packets (ID 25, 28) are not yet received. RWS, LAF, and LFR represent Receive Window Size, Largest Acceptable Frame, Largest Frame Received, respectively. RWS should be less than equal to the difference of LAF and LFR (i.e., LAF – LFR \( \geq \) RWS) as shown in Figure 8.2.

![Figure 8.1: Sender window window size](image-url)
The working of Sliding Window has been depicted in Figure 8.3 and Figure 8.4. The sender continues to transmit data frames and the sending window will stay at LFS till ACK received for all frames sent. The receiver transmits ACK after processing each data frame. Upon sending/receiving ACK in sequence, the receiver/sender window moves to the next sequence number as shown in Figure 8.5.
Moreover, the sender also retransmits frame for which it could not receive ACK.

![Image of sliding window working – out-of-order ACK receiving]

**Figure 8.5:** Sliding window working – out-of-order ACK receiving

### 8.3 Implementation of Sliding Window Algorithm

In following simulation, we are going to present the sliding window algorithm in which packets are delivered at the destination (Application Layer) in an in-order fashion. We will implement this algorithm by using unique sequence number series assignment packets. This simulation will ensure that this sequence is maintained at the destination to ensure the in-order delivery. To achieve reliability, we will utilize ACK for each successfully received packet, but we will also ensure that no out of order packet will be forward to the upper layer. To understand Sliding Window through simulation in a Point-to-Point network, steps details have been mentioned below:

1- To create OMNeT++ project, follow steps (1—9) of section 4.2.
2- To create network description files (.ned), follow steps (2—14) of section 5.4.
3- To create TCP/IP packet, follow steps (15—17) of section 5.4. Write code shown in Listing 8.1 for `packet.msg` and to create respective implementation files (.cc), follow step 19 of section 5.4.
4- Write packet attributes and save as `packet.msg` as shown in Listing 8.1.

#### Listing 8.1: Attributes of TcpIpPacket packet

```c
Line 1: packet TcpIpPacket{
Line 2:     string data = NULL;
Line 3:     string appHeader = NULL;
Line 4:     string tcpHeader = NULL;
Line 5:     string ipHeader = NULL;
Line 6:     string dllHeader = NULL;
Line 7:     string phyHeader = NULL;
Line 8:     int sequenceNo = 0;
Line 9: }
```

In Listing 8.1, `TcpIpPacket` is created with headers of each layer, integer type sequence number (assign to each generated packets at the Application layer), and data.

5- Write code for `ACK_packet.msg` as shown in Listing 8.2.
An acknowledgement packet is created in Listing 8.2 which will be used at Data Link Layer.

6- Write C++ code for ApplicationLayer.cc as shown in Listing 8.3.

Listing 8.2
Attributes of Acknowledgement packet

| Line 1: | packet ackPacket{ |
| Line 2: | bool ack = false; |
| Line 3: | int sequenceNo = 0; |
| Line 4: | }

Listing 8.3
Implementation of ApplicationLayer module

| Line 1: | #include<stdio.h> |
| Line 2: | #include<string.h> |
| Line 3: | #include<omnetpp.h> |
| Line 4: | #include"packet_m.h" |
| Line 5: | using namespace omnetpp; |
| Line 6: | class ApplicationLayer : public cSimpleModule{ |
| Line 7: | private: |
| Line 8: | int kind = 0; |
| Line 9: | int Sequence = 0; |
| Line 10: | protected: |
| Line 11: | virtual void initialize() override; |
| Line 12: | virtual void handleMessage(cMessage *msg) override; |
| Line 13: | virtual TcpIpPacket* generateNextMessage(); |
| Line 14: | }; |
| Line 15: | Define_Module(ApplicationLayer); |
| Line 16: | void ApplicationLayer::initialize(){ |
| Line 17: | if(strcmp(par("AppID").operator const char *(),"PC1")==0){ |
| Line 18: | scheduleAt (0.0, generateNextMessage()); |
| Line 19: | } |
| Line 20: | } |
| Line 21: | void ApplicationLayer::handleMessage(cMessage *msg){ |
| Line 22: | TcpIpPacket* pkt = check_and_cast<TcpIpPacket *>(msg); |
| Line 23: | if((pkt->arrivedOn ("appGate$i",0)){ |
| Line 24: | bubble ((pkt->getData()); |
| Line 25: | delete pkt; |
| Line 26: | } |
| Line 27: | if(msg->isSelfMessage()){ |
| Line 28: | //Send message packet to Transport Layer |
| Line 29: | send(pkt,"appGate$o",0); |
| Line 30: | //Generate next message after 100ms |
| Line 31: | scheduleAt (simTime()+.1, generateNextMessage()); |
| Line 32: | } |
| Line 33: | //Generate new packet with specific format |
| Line 34: | TcpIpPacket* ApplicationLayer::generateNextMessage(){ |
| Line 35: | char packetInfo [20]; |
| Line 36: | sprintf(packetInfo,"Message %d",Sequence); |
/* Use of kind variable to create data packet with five different colors to differentiate each packet in GUI */

Line 36:    TcpIpPacket* nextPkt = new TcpIpPacket(packetInfo, kind++ % 5);
Line 37:    if(kind == 5)    kind = 0;
Line 38:    if(kind == 5)    kind = 0;

// To range sequence number from 0 to 9
Line 41:    nextPkt->setSequenceNo(Sequence%10);
Line 42:    nextPkt->setData(packetInfo);

// To increment in sequence number
Line 43:    Sequence++;    return nextPkt;

At initializing phase (lines 16—20 of Listing 8.3), network restricts itself to initiate packet transmission from the Application layer having AppID = PC1 using comparison through operator const char*(). AppID is defined in omnetpp.ini (Listing 8.8). At handling phase (line 21 to 45 of Listing 8.3), the packet generated by the Application layer (line 33 of Listing 8.3). The generateNextMessage() function returns next generated packet (line 44) with unique sequence number assignment (line 41) and color (line 36). Sequence numbers will be utilized at the Data Link layer for implementation of sliding window algorithm. At handling phase (lines 21—32 of Listing 8.3), the Application layer checks whether it has received the packet from the Transport layer or it is a self-message. Packet arrival from the Transport layer (line 23) indicates the successful packet reception. On the other hand, in the case of self-message (line 27), the Application layer will send generated a message to the Transport layer (line 28). Each generated packet will be transmitted after 100 ms (line 29).

Write C++ code for TransportLayer.cc as shown in Listing 8.4.

Listing 8.4 Implementation of TransportLayer module

Line 1:   #include<stdio.h>
Line 2:   #include<string.h>
Line 3:   #include<omnetpp.h>
Line 4:   #include<packet_m.h>
Line 5:   using namespace omnetpp;

Line 7:   class TransportLayer : public cSimpleModule{
Line 8:   protected:
Line 9:       virtual void initialize() override;
Line 10:      virtual void handleMessage(cMessage *msg) override;
Line 11:   };

Line 12:   Define_Module(TransportLayer);

Line 13:   void TransportLayer::initialize()
Line 14:   {
Line 15:       TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
Line 16:       if(pkt->arrivedOn("tcpGate$i",0))
Line 17:           send(pkt, "tcpGate$o",1);
At handling phase (lines 14—22 of Listing 8.4), Transport layer checks whether it received a packet from Application layer or Network layer. Upon receiving a packet from Network layer (line 19), it forwards to the Application layer (line 20). If it received a packet from the Application layer (line 16), then it forwards this packet to the Network layer (line 17).

8- Write C++ code for NetworkLayer.cc as shown in Listing 8.5.

```
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include <packet_m.h>
#include <vector>
using namespace omnetpp;

class NetworkLayer : public cSimpleModule{
protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(NetworkLayer);

void NetworkLayer::initialize()
{
    // Receive packet from Network Layer
    if(pkt->arrivedOn("tcpGate$i",1)){
        // Send packet Application
        send(pkt, "tcpGate$o", 0);
    }
}

void NetworkLayer::handleMessage(cMessage *msg)
{
    TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
    // Receive packet from Transport layer
    if(pkt->arrivedOn("ipGate$i",0)){
        // Send packet to Data Link layer
        send(pkt, "ipGate$o", 1);
    }
    // Receive packet from Data Link layer

    if(pkt->arrivedOn("ipGate$i",1)){
        // Send packet to Transport layer
        send(pkt, "ipGate$o", 0);
    }
}
```

At handling phase (lines 14—23 of Listing 8.5), Network layer checks whether it receives a packet from the Data Link layer or the Transport layer. Upon receiving a packet from the Data Link layer (line 19), it forwards the packet to the Transport layer (line 20). If received packet from the Transport layer (line 16), it forwards it to Data Link layer (line 17).

9- Write C++ code for DataLinkLayer.cc as shown in Listing 8.6.
Listing 8.6 Implementation of DataLinkLayer module

Line 1: #include<stdio.h>
Line 2: #include<string.h>
Line 3: #include<omnetpp.h>
Line 4: #include<packet_m.h>
Line 5: #include<ACK_Packet_m.h>
Line 6: using namespace omnetpp;

Line 7: class DataLinkLayer : public cSimpleModule{
Line 8:  cQueue packetsQueue; //Queue for in-order packet delivery
Line 9:  cPar* macAddress; //For destination MAC address
Line 10:     //For available position in source sliding window
Line 11:  bool sourceSlidingWindowFlags[5];
Line 12:     //For available position in destination sliding window
Line 13:  bool destinationSlidingWindowFlags[5];
Line 14:  int sequenceNo = 0; //For unique packet ID
Line 15:  TcpIpPacket *pkt;
Line 16:  TcpIpPacket **sourceSlidingWindow; //Source sliding window
Line 17:  TcpIpPacket **destinationSlidingWindow; //Receive sliding window
Line 18:     //To access index of source sliding window
Line 19:     int pointerLocationSourceSW = 0;
Line 20:     //To access index of destination sliding window
Line 21:     int pointerLocationDestinationSW = 0;

Line 22: public: DataLinkLayer(){
Line 23:     /* To create Sender and Receiver Sliding windows of size 5 and respective flag values (false means vacant positions) */
Line 24:     for(int i=0;i<5;i++) {
Line 25:         sourceSlidingWindow = new TcpIpPacket*[i];
Line 26:         destinationSlidingWindow = new TcpIpPacket*[i];
Line 27:         sourceSlidingWindowFlags[i] = false;
Line 28:         destinationSlidingWindowFlags[i] = false;
Line 29:     }
Line 30:     pkt = new TcpIpPacket;
Line 31: }

Line 32: ~DataLinkLayer(){
Line 33:     for(int i=0;i<5;i++) {
Line 34:         delete sourceSlidingWindow[i];
Line 35:         delete destinationSlidingWindow[i];
Line 36:     }
Line 37:     delete pkt;
Line 38: }

Line 39: protected:
Line 40:     virtual void initialize() override {
Line 41:         //Get MAC address declared in omnetpp.ini
Line 42:         macAddress = &par("macAddress");
Line 43:     }

Line 44:     virtual void handleMessage(cMessage *msg) override {
Line 45:         //....
Line 46:     };

Line 47: Define_Module(DataLinkLayer);
// Packet arrived from Network layer
if (msg->arrivedOn("dllGate$i", 0)) {
    // Assign Sequence Number to received packets
    char sequenceNoInfo[10];
sprintf(sequenceNoInfo, "seq# %d", sequenceNo);
    sequenceNo = sequenceNo%10; sequenceNo++;
    msg->setName(sequenceNoInfo);
    // Push packet in packet queue
    packetsQueue.insert(msg);
}
}

// Receive packet from Physical layer
if (msg->arrivedOn("dllGate$i", 1)) {
    /* To match with Receiver MAC address. It shows received packet from Physical layer of destination node */
    if (strcmp("AC-F4-BB-7B-57-51", macAddress->stringValue()) == 0) {
        TcpIpPacket* receivedPacket = check_and_cast<TcpIpPacket*>(msg);
        /* Compare MAC Address of destination node with MAC address incorporated in received packet */
        if (strcmp(receivedPacket->getDllHeader(), macAddress->stringValue()) == 0) {
            /* To check destinationSlidingWindowFlags is False at mentioned Index */
            if (!destinationSlidingWindowFlags[receivedPacket->getSequenceNo()%5]) {
                /* To make index position True(to ensure packet is properly received) */
                destinationSlidingWindowFlags[receivedPacket->getSequenceNo()%5] = true;
                /* Save received packet at same index value of Destination Sliding Window */
                destinationSlidingWindow[receivedPacket->getSequenceNo()%5] = receivedPacket->dup();
            } /* Generate ACK for sender with same Sequence No. */
            char ackMessageToSource[20];
sprintf(ackMessageToSource, "ACK seq# %d", receivedPacket->getSequenceNo());
            ackPacket *ackToSource = new ackPacket (ackMessageToSource, 6);
            ackToSource->setSequenceNo(receivedPacket->getSequenceNo());
            // Send ACK packet to Physical Layer
            send(ackToSource, "dllGate$o", 1);
            /* For successful reception of Data Packet */
            if (destinationSlidingWindowFlags[pointerLocationDestinationSW] == true) {
                /* To set destinationSlidingWindowFlags to false to Move Sliding window */
                destinationSlidingWindowFlags[pointerLocationDestinationSW] = false;
                /* To make copy of received packet */
                receivedPacket = destinationSlidingWindow[pointerLocationDestinationSW]->dup();
                // Send received packet to Network layer
                send(receivedPacket, "dllGate$o", 0);
            }
        }
    }
}

// To move Sliding Window forward
Line 66:    pointerLocationDestinationSW = (pointerLocationDestinationSW + 1) % 5;

Line 67: }
/* If device MAC doesn't match to Data Link layer header (packet drop scenario) */
Line 68: } else {
Line 69:    bubble("packet drop");
Line 70:    delete receivedPacket;

Line 71: }
/* Received packet from Physical layer of source node (where only reception of ACK is possible */
Line 72: else {
Line 73:    ackPacket * ackFromDestination = check_and_cast<ackPacket *>(msg);
    /* For successful ACK received make Flag false for same Sequence No. index */
Line 74:    sourceSlidingWindowFlags[ackFromDestination->getSequenceNo() % 5] = false;
    /* To cancel re-transmission because of successful in-time ACK reception */
Line 75:    cancelEvent(sourceSlidingWindow[ackFromDestination->getSequenceNo() % 5]);

Line 76: }
// To manage Source Sliding Window
Line 77: if(!packetsQueue.isEmpty() && !msg->isSelfMessage()){
    /* Check sourceSlidingWindowFlag = False at source sliding window pointer index */
Line 78:    if(sourceSlidingWindowFlags[pointerLocationSourceSW] == false){
        // To protect from any conflict
Line 79:        sourceSlidingWindowFlags[pointerLocationSourceSW] = true;
        // To pop the data packet from queue
Line 80:        pkt = check_and_cast<TcpIpPacket *>(packetsQueue.pop());
        /* Copy data packet at selected index (source sliding window pointer) */
Line 81:        sourceSlidingWindow[pointerLocationSourceSW] = pkt->dup();
        /* To generate packet drop scenario, use random assignment of MAC address to drop packet */
Line 82:        int randomDestinationMac = uniform(1,3);
Line 83:        if(randomDestinationMac == 2)
Line 84:            pkt->setDllHeader("AC-F4-BB-7B-57-51");
            // To set destination MAC address
Line 85:        else
Line 86:            pkt->setDllHeader("NULL");
        // To send Data Packet at Physical layer of sender node
Line 87:        send(pkt,"dllGate$o",1);
        /* Re-schedule same Packet after 210ms. If ACK of packet is receives in-time, then this event will be cancelled */
Line 88:        scheduleAt(simTime() + 0.21,
Line 89:            sourceSlidingWindow[pointerLocationSourceSW]);
// Move Sender sliding window pointer Forward
Line 90:    }
// Re-send packet scenario (ACK not received in-time)
Line 91:    if(msg->isSelfMessage()){
Line 92:        TcpIpPacket* resendPacket = new TcpIpPacket;
Line 93:    resendPacket = check_and_cast<TcpIpPacket*>(msg);
        /* To copy packet in specific sequence no. index of Source Sliding Window */
Line 94:    sourceSlidingWindow[resequenceNumber->
        getSequenceNo()%5]=resequenceNumber->dup();
        /* Make Flags true to reserve slot till successful ACK received */
Line 95:    sourceSlidingWindowFlags [resequenceNumber->
        getSequenceNo()%5]=true;
        /* Re-scheduled packet if ACK of this packet receive in-time
        then cancel this event */
Line 96:    scheduleAt (simTime()+ 0.21,
        sourceSlidingWindow[resequenceNumber->
        getSequenceNo()%5]);
    //Use for random assignment of MAC to drop packet
Line 97:    int randomDestinationMac =uniform(1,3);
Line 98:    if(randomDestinationMac==2)
        //To set MAC address of destination
Line 99:        resendPacket->setDllHeader("AC-F4-BB-7B-57-51");
Line 100:    else
Line 101:        resendPacket->setDllHeader("NULL");
    //To re-send packet to Physical layer of sender node
Line 102:        send(resendPacket,"dllGate$o",1);}
Line 103:    }

At constructor, source and destination sliding windows and their corresponding flags are initialized (lines 18—26 of Listing 8.6). At handling phase (lines 40—103 of Listing 8.6), Data Link layer checks whether it received a packet from Physical layer or Network layer. Upon receiving a packet from Network layer of the source node (line 42 of Listing 8.6), Data Link layer maintains and assigns the message sequence number (line 43—46 of Listing 8.6) and at the same time packet will be inserted in packet queue (line 47 of Listing 8.6). Upon receiving packet from Physical layer of the destination node (line 50 of Listing 8.6, index (corresponding to received sequence number) of destination sliding window is checked through its flag (line 53 of Listing 8.6). If index location is available (i.e., flag = false), then that index flag is assigned true (line 54 of Listing 8.6). A copy of received packet is placed at selected index of the sliding window (line 55 of Listing 8.6). An acknowledgement packet is created with the same sequence number (lines 56—59 of Listing 8.6) and sent to the source node (line 61 of Listing 8.6). On the other hand, if index (corresponding to received sequence number) of destination sliding window is true which means the successful reception of the packet, then make current location flag = false (line 63 of Listing 8.6). Make location vacant for next packet and forward packet to the Network layer (line 65 of Listing 8.6). Move the sliding window (line 66 of Listing 8.6). If the packet is dropped (line 68 of Listing 8.6), then there will be no activity on receiving end. Upon receiving packet from Network layer of the source node (line 42), the sequence number is assigned (lines 43—46) to received packet and push packet to the queue (line 47). The source node is responsible for managing and processing the packets in the queue. The packet is popped from the queue (line 80 of Listing 8.6) and a copy of the popped packet is placed at available sliding window index. At lines 82—86, error introduced in a random way by assigning wrong MAC address and that packet is transmitted to the destination node through physical layer (line 87 of Listing 8.6). Packet retransmission is scheduled after 0.21 sec (line 88 of Listing 8.6) and pointer of source sliding window is incremented. If scheduled event is not canceled before 0.21 sec (means acknowledgment not received), then packet retransmission is required (line 91 of Listing 8.6) and sliding window desired index will keep reserve by making its flag = True (line 95 of Listing 8.6). The packet is retransmitted to Physical layer of the source node (line 102 of Listing 8.6).

10- Write C++ code for PhysicalLayer.cc as shown in Listing 8.7.
Listing 8.7  Implementation of PhysicalLayer module

Line 1: #include<stdio.h>
Line 2: #include<string.h>
Line 3: #include<omnetpp.h>
Line 4: #include<packet_m.h>
Line 5: using namespace omnetpp;

Line 6: class PhysicalLayer : public cSimpleModule{
Line 7: protected:
Line 8: virtual void initialize() override;
Line 9: virtual void handleMessage(cMessage *msg) override;
Line 10: };

Line 11: Define_Module(PhysicalLayer);

Line 12: void PhysicalLayer::initialize(){

Line 13: void PhysicalLayer::handleMessage(cMessage *msg){
    // Receive packet from Data Link layer
    if(msg->arrivedOn("phyGate$i",0)){
        //Send packet to the channel
        send(msg, "phyGate$o",1);
    }
    else if(msg->arrivedOn("phyGate$i",1)){
        //Send packet to Data Link layer
        send(msg, "phyGate$o",0);
    }
}

Line 20: }

At handling phase (line 13—20 of Listing 8.7), Physical layer checks whether it receives a packet from Data Link layer or Physical layer of connected PC. Upon receiving a packet from Physical layer of connected PC (line 17), the packet is forwarded to Data Link layer (line 18) and if receives a packet from Data Link layer (line 14), it is re-transmitted towards connected PC (line 15).

11- To create a network initialization file (omnetpp.ini), right click on the OMNET++_Sliding_window_working folder and then select New → Initialization File (.ini).
12- Select OMNET++_Sliding_window_working and write file name as omnetpp.ini (if not already written) in File name text field and proceed with Next button. Empty Ini File in Select template text area and press Next.
13- Press browse button to select network in New Ini File window.
14- From available list, select NED type related to your project. The selected NED type will appear in the text field. Click Finish to complete the creation of omnetpp.ini.
15- Write the omnetpp.ini file as shown in Listing 8.8.

Listing 8.8  Contents of omnetpp.ini

Line 1: [General]
Line 2: network = omnet_sliding_window_working.NEDSrc.Pt2Pt_Network
Line 3: **.PC1.d1L.macAddress = "EA-E3-AB-3A-27-5F"
Line 4: **.PC2.d1L.macAddress = "AC-F4-BB-7B-57-51"
Line 5: **.PC1.appL.AppID = "PC1"
Line 6: **.PC2.appL.AppID = "PC2"
The project files can be seen in Project Explorer (Figure 8.6).

![Figure 8.6: View of project files in project explorer window](image)

In this simulation, we presented the in-order delivery of packets from source to destination. We have simulation results providing the reception of packets on application layer with a patterned sequence number and acknowledge of each packet sent to the source node.

**Practice Task**

1. Create a network having three nodes as shown in Figure 8.7. Both PC1 and PC2 nodes are required to send packets (PKT) to each other via intermediate node R. Sliding window algorithm is required to be implemented with \( SWS = RWS = 3 \). The simulation must show that if \( PKT[6] \) is in the receive window, then \( PKT[0] \) (or any older data) should not arrive at the receiver. Similarly, simulate that if \( ACK[6] \) is sent, then \( ACK[2] \) (or any earlier ACK) cannot be received.

![Figure 8.7: Network topology for practice task 1](image)
Implementation of Multiplexing/Demultiplexing

Learning Objectives:

After studying this chapter, you will be able to describe:

- the concept of multiplexing/demultiplexing in TCP/IP Stack
- implementation of multiplexing/demultiplexing in TCP/IP Stack in OMNeT++
9.1 Multiplexing and Demultiplexing in TCP/IP Stack

In computer networks, information exchange takes place between matching applications running on various devices. TCP/IP application instances (also known as processes) use TCP/IP protocol stack for sending and receiving packets either through User Datagram Packet (UDP) or Transmission Control Protocol (TCP). Concerning multiple processes on a typical TCP/IP host, multiplexing and demultiplexing at the Transport layer is related to the extension of host-to-host delivery service to process-to-process delivery service. Upon receiving packets from the Network layer, the Transport layer takes responsibility for packet delivery to the appropriate application process as shown in Figure 9.1. Figure 9.1 depicts that at the Transport layer, TCP and UDP may receive data packets from multiple application processes and transmit all to the IP layer and vice versa.

![Figure 9.1: Multiplexing/Demultiplexing at transport layer](image)

In network terminology, multiplexing and demultiplexing are possible through sockets (data entry points of an application to the network). In short, data collection from multiple sockets and data delivery to correct socket is known as multiplexing and demultiplexing, respectively.

9.2 Implementation of Multiplexing/Demultiplexing

To implement the concept of Multiplexing and Demultiplexing within TCP/IP protocol stack in a Point-to-Point network, step details have been mentioned below:

1- To create a project related to multiplexing/de-multiplexing, follow steps (1—9) of section 4.2.

![Figure 9.2: Viewing of the NED files in project explorer window](image)
To create twelve NED files (i.e., PhysicalLayer.ned, DataLinkLayer.ned, NetworkLayer.ned, TransportLayer.ned, ApplicationLayer.ned, UserApplication.ned, FTP_Protocol.ned, HTTP_Protocol.ned, TFTP_Protocol.ned, TCP.ned, UDP.ned, MuxDemuxMudule.ned) follow steps (2—14) of section 5.4. All created .ned files will appear within NEDSrc folder in Project Explorer window (Figure 9.2).

3- Write code in the UserApplication.ned as shown in Listing 9.1.

### Listing 9.1 Source view of UserApplication NED

```plaintext
Line 1: package tcpipstack3.NEDSrc;
Line 2:  simple UserApplication{
Line 3:    parameters:
Line 4:      string userID;
Line 5:    gates:
Line 6:      inout userAppGate[3];
Line 7: }
```

4- Write code in the HTTP_Protocol.ned as shown in Listing 9.2.

### Listing 9.2 Implementation of HttpProtocol NED

```plaintext
Line 1: package tcpipstack3.NEDSrc;
Line 2:  simple HttpProtocol{
Line 3:    gates:
Line 4:      inout httpGate[2];
Line 5: }
```

5- Write code in the TFTP_Protocol.ned as shown in Listing 9.3.

### Listing 9.3 Source view of TftpProtocol NED

```plaintext
Line 1: package tcpipstack3.NEDSrc;
Line 2:  simple TftpProtocol{
Line 3:    gates:
Line 4:      inout tftpGate[2];
Line 5: }
```

6- Write code in the FTP_Protocol.ned as shown in Listing 9.4.

### Listing 9.4 Source view of FtpProtocol NED

```plaintext
Line 1: package tcpipstack3.NEDSrc;
Line 2:  simple FtpProtocol{
Line 3:    gates:
Line 4:      inout ftpGate[2];
Line 5: }
```

7- Write code in Application.ned as shown in Listing 9.5.

### Listing 9.5 ApplicationLayer module

```plaintext
Line 1: package tcpipstack3.NEDSrc;
Line 2:  module Application{
Line 3:    parameters:
Line 4:      @display("i=device/pc2,gold;bgl=2");
```
In Listing 9.5, the Application layer is designed as a compound module (which is required to assign a port number). There are three inout gates of the Application layer (line 6 of Listing 9.5). Four already created simple modules (i.e., FTP, HTTP, TFTP, and UserApplication) are submodules of this compound module and have three internal connections (line 21—23 of Listing 9.5) and three external connections (line 24—26 of Listing 9.5).

8- Write code in the MuxDemuxModule.ned as shown in Listing 9.6.

Listing 9.6 Source view of MuxDemuxModule NED

```
package tcpipstack3.NEDSrc;

simple MuxDemuxModule{
    gates:
        inout muxDemuxGate[3];
}
```

9- Write code in the TCP.ned as shown in Listing 9.7.

Listing 9.7 Source view of Tcp NED

```
package tcpipstack3.NEDSrc;

simple Tcp{
    gates:
        inout tcpGate[3];
}
```

10- Write code in the UDP.ned as shown in Listing 9.8.

Listing 9.8 Source view of Udp NED

```
package tcpipstack3.NEDSrc;

simple Udp{
```
11- Write code in the *TransportLayer.ned* as shown in Listing 9.9.

```ned
package tcpipstack3.NEDSrc;
module TransportLayer{
    parameters:
    @display("p=123,56;i=block/filter");
    gates:
    inout transGate[4];
    submodules:
    tcp: Tcp {
        @display("p=123,137;i=old/proc2");
    }
    udp: Udp {
        @display("p=123,216;i=device/card");
    }
    muxDmux: MuxDemuxModule {
        @display("p=223,181;i=block/join");
    }
    connections:
    muxDmux.muxDemuxGate[0] <-> tcp.tcpGate[2];
    muxDmux.muxDemuxGate[1] <-> udp.udpGate[1];
    muxDmux.muxDemuxGate[2] <-> transGate[3];
    tcp.tcpGate[0] <-> transGate[0];
    tcp.tcpGate[1] <-> transGate[1];
    udp.udpGate[0] <-> transGate[2];
}
```

In Listing 9.9, the Transport layer is designed as a compound module to explain the process of multiplexing and de-multiplexing. There are four *inout* gates of the Transport layer (line 6 of Listing 9.9). Three simple modules *muxDmux, tcp*, and *udp* are submodules of this module and have three internal connections (Listing 18—20 of Listing 9.9) and four external connections (line 21—23 of Listing 9.9).

12- Write code in the *NetworkLayer.ned* as shown in Listing 9.10.

```ned
package tcpipstack3.NEDSrc;
simple NetworkLayer{
    parameters:
    string ipAddress = "NIL";
    gates:
    inout ipGate[2];
}
```

13- Write code in the *DataLinkLayer.ned* as shown in Listing 9.11.

```ned
package tcpipstack3.NEDSrc;
```
Line 2:    simple DataLinkLayer{
Line 3:        parameters:
Line 4:            string macAddress = "NIL";
Line 5:        gates:
Line 6:            inout dllGate[2];
Line 7:    }

14-    Write code in the PhysicalLayer.ned as shown in Listing 9.12.

Listing 9.12 [Source view of PhysicalLayer NED]

Line 1:    package tcpipstack3.NEDSrc;
Line 2:    simple PhysicalLayer{
Line 3:        gates:
Line 4:            inout phyGate[2];
Line 5:    }

15-    Create a compound modules named PC.ned and write code as shown in Listing 9.13.

Listing 9.13 [Implementation of PC module and network description in NED]

    // Package statement
Line 1:    package tcpipstack3.NEDSrc;
    // A compound module creation named PC with inout gate vector
Line 2:    module PC{
Line 3:        parameters:
Line 4:            @display("i=device/pc2,gold");
Line 5:        gates:
Line 6:            inout gate[];
    // Inclusion of Layers as submodules in PC module
Line 7:        submodules:
Line 8:            appL: ApplicationLayer {
Line 9:                @display("p=45,26;i=block/sink");
Line 10:            }
Line 11:            tcpL: TransportLayer {
Line 12:                @display("p=123,56;i=block/filter");
Line 13:            }
Line 14:            ipL: NetworkLayer {
Line 15:                @display("p=123,137;i=old/proc2");
Line 16:            }
Line 17:            dlL: DataLinkLayer {
Line 18:                @display("p=123,216;i=device/card");
Line 19:            }
Line 20:            phyL: PhysicalLayer {
Line 21:                @display("p=57,226;i=device/card");
Line 22:            }
Line 23:        connections:
Line 24:            appL.appGate[0] <-> tcpL.tcpGate[0];
Line 25:            tcpL.tcpGate[1] <-> ipL.ipGate[0];
Line 26:            ipL.ipGate[1] <-> dlL.dllGate[0];
Line 27:            dlL.dllGate[1] <-> phyL.phyGate[0];
Line 28:            phyL.phyGate[1] <-> gate++;
Line 29:    }

16-    To create TCP/IP packet, follow steps (15—17) of section 5.4. Write code shown in Listing 9.14 in packet.msg. Write following packet attributes and save as packet.msg.
Listing 9.14  Attributes of data packet

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>struct applicationPacket{</code></td>
</tr>
<tr>
<td>2</td>
<td><code>int data;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>int protocol;</code></td>
</tr>
<tr>
<td>4</td>
<td><code>int ip;</code></td>
</tr>
<tr>
<td>5</td>
<td><code>int portNo;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>};</code></td>
</tr>
<tr>
<td>7</td>
<td><code>packet TcpIpPacket {</code></td>
</tr>
<tr>
<td>8</td>
<td><code>int srcAdd;</code></td>
</tr>
<tr>
<td>9</td>
<td><code>int destAdd;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>applicationPacket appHeader;</code></td>
</tr>
<tr>
<td>11</td>
<td><code>int tcpHeader;</code></td>
</tr>
<tr>
<td>12</td>
<td><code>string ipHeader;</code></td>
</tr>
<tr>
<td>13</td>
<td><code>string dllHeader;</code></td>
</tr>
<tr>
<td>14</td>
<td><code>string phyHeader;</code></td>
</tr>
<tr>
<td>15</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

TcpIpPacket is created with all corresponding headers for each layer. Additionally, an application sub-packet is also created inside the TcpIpPacket which contain the data, protocol, ip address, and portNo of the destination node (lines 1—6 of Listing 9.14). Selection of integer type values for application sub-packet is done to make computation simpler at all layers.

17- To create respective implementation files (.cc for each .ned file created in step 2 of section 9.2), follow step 19 of section 5.4.
18- Write C++ code for UserApplicationLayer.cc as shown in Listing 9.15.

Listing 9.15  Implementation of UserApplication module

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>#include&lt;stdio.h&gt;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>#include&lt;string.h&gt;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>#include&lt;omnetpp.h&gt;</code></td>
</tr>
<tr>
<td>4</td>
<td><code>#include&quot;packet_m.h&quot;</code></td>
</tr>
<tr>
<td>5</td>
<td><code>using namespace omnetpp;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>class UserApplication : public cSimpleModule{</code></td>
</tr>
<tr>
<td>7</td>
<td><code>protected:</code></td>
</tr>
<tr>
<td>8</td>
<td><code>virtual void initialize() override ;</code></td>
</tr>
<tr>
<td>9</td>
<td><code>virtual void handleMessage(cMessage *msg) override;</code></td>
</tr>
<tr>
<td>10</td>
<td><code>virtual TcpIpPacket* generateNextMessage();</code></td>
</tr>
<tr>
<td>11</td>
<td><code>};</code></td>
</tr>
<tr>
<td>12</td>
<td><code>Define_Module(UserApplication);</code></td>
</tr>
<tr>
<td>13</td>
<td><code>void UserApplication::initialize(){</code></td>
</tr>
<tr>
<td>14</td>
<td><code>/To send first message from PC1</code></td>
</tr>
<tr>
<td>15</td>
<td><code>if(strcmp(par(&quot;userID&quot;).operator const char *(),&quot;PC1&quot;) == 0){</code></td>
</tr>
<tr>
<td>16</td>
<td><code>scheduleAt (0.0,generateNextMessage());</code></td>
</tr>
<tr>
<td>17</td>
<td><code>}</code></td>
</tr>
<tr>
<td>18</td>
<td><code>void UserApplication::handleMessage(cMessage *msg){</code></td>
</tr>
<tr>
<td>19</td>
<td><code>TcpIpPacket* pkt = check_and_cast&lt;TcpIpPacket*&gt;(msg);</code></td>
</tr>
<tr>
<td>20</td>
<td>`if(pkt-&gt;arrivedOn(&quot;userAppGate$i&quot;,0)</td>
</tr>
<tr>
<td></td>
<td>`pkt-&gt;arrivedOn(&quot;userAppGate$i&quot;,1)</td>
</tr>
</tbody>
</table>
pkt->arrivedOn("userAppGate$i",2) }
Line 21:    if(pkt->getAppHeader().protocol==1)
Line 22:        bubble("data of http protocol received");
Line 23:    else if(pkt->getAppHeader().protocol==2)
Line 24:        bubble("data of htp protocol received");
Line 25:    else if(pkt->getAppHeader().protocol==3)
Line 26:        bubble("data of thtp protocol received");
Line 27:    delete pkt;
Line 28: }

// Self-message for initialization or resending packet
Line 29:    if(msg->isSelfMessage()){
Line 30:        //1 is considered as HTTP protocol
Line 31:            send(pkt,"userAppGate$o",0);
Line 32:        //2 is considered as FTP protocol
Line 33:            send(pkt,"userAppGate$o",1);
Line 34:        //3 is considered as TFTP protocol
Line 35:            send(pkt,"userAppGate$o",2);
Line 36:            scheduleAt(simTime()+.1, generateNextMessage());
Line 37:    }
Line 38: }

TcpIpPacket* UserApplication::generateNextMessage(){
    char packetInfo[200];
    int message = uniform(6,9);
    int protocol = uniform(0,4);
    applicationPacket appHdr;
    appHdr.data = message;
    appHdr.ip = 192168641;
    appHdr.protocol = protocol;
    sprintf(packetInfo,"Data:%d",message);
    TcpIpPacket* nextPkt = new TcpIpPacket(packetInfo);
    nextPkt->setAppHeader(appHdr);
    return nextPkt;
Line 51: }

At initialization phase (lines 13—17 of Listing 9.15), generated packet is scheduled at PC1. Application packet is created using generateMessage() function (lines 39—51 of Listing 9.15). The generated packet contains data, IP address, and application protocol (lines 44—46 of Listing 9.15).

Three application processes at an abstract level (named as HTTP, FTP, TFTP) have been considered for understanding the concept of multiplexing and demultiplexing (by assigning integer protocol values (Line 42 of Listing 9.15). If protocol value is 1, then it is Http module packet. This packet will be sent to gate 0, which is further connected to the Tcp module. If protocol value is 2, then it is Ftp module. This packet will be sent to gate 1, which is also connected to the Tcp module. If protocol value is 3, then it is a Tftp module packet. This packet will be sent to gate 3, which is further connected to Tcp module.

19- Write C++ code for ftp.cc as shown in Listing 9.16.
At handling phase (lines 13—29 of Listing 9.16), upon packet reception from the Application layer at the input port of Ftp module (line 19 of Listing 9.16), the port number is assigned and set in the Application header (lines 20—21 of Listing 9.16). This is followed by packet transmission to the output port (line 23 of Listing 9.16). On the other hand, if the packet is received on the input port of Ftp module, then it will be sent to the output port of Ftp module which is connected with the UserApplication module.

20- Write C++ code for http.cc as shown in Listing 9.17.
virtual void handleMessage(cMessage *msg) override;
};

Define_Module(HttpProtocol);

void HttpProtocol::initialize(){
  
  void HttpProtocol::handleMessage(cMessage *msg){
    
    TcpIpPacket* pkt = check_and_cast<TcpIpPacket*>(msg);
    applicationPacket appPacket;
    appPacket.ip = pkt->getAppHeader().ip;
    appPacket.data = pkt->getAppHeader().data;
    appPacket.protocol = pkt->getAppHeader().protocol;
    if (pkt->arrivedOn("httpGate$i", 0)){
      appPacket.portNo = 80;
      pkt->setAppHeader(appPacket);
      bubble("Set port 80");
      send(pkt,"httpGate$o",1);
    }
    // Self-message for initialization or re-sending scenario
    if (pkt->arrivedOn("httpGate$i", 1)){
      bubble("http protocol data received: port 80");
      send(pkt,"httpGate$o",0);
    }
  }
  }

At handling phase (line 13—29 of Listing 9.17), upon packet receive at the input port of
Http module (line 19 of Listing 9.17), the port number is assigned and attached to the application
packet (lines 20—21). The application packet will be sent to the output port (line 23 of Listing 9.17)
which is connected to the Application layer module. On the other hand, if the packet received at
the input port of Http module, it is transmitted to the output port of Http module.

21- Write C++ code for tftp.cc as shown in Listing 9.18.

Listing 9.18 Implementation of TftpProtocol module

#include<stdio.h>
#include<string.h>
#include<omnetpp.h>
#include"packet_m.h"
using namespace omnetpp;

class TftpProtocol:public cSimpleModule{
protected:
  virtual void initialize()override;
  virtual void handleMessage(cMessage *msg) override;
};

Define_Module(TftpProtocol);

void TftpProtocol::initialize(){
  
  void TftpProtocol::handleMessage(cMessage *msg){
    
    TcpIpPacket* pkt = check_and_cast<TcpIpPacket*>(msg);
    applicationPacket appPacket;
    appPacket.ip = pkt->getAppHeader().ip;
At handling phase (line 13—29 of Listing 9.18), upon packet received at the input port of Tftp module (line 19 of Listing 9.18), the port number is assigned and attached to the application packet (lines 20—21). The application packet will be sent to the output port (line 23 of Listing 9.18) which is connected to the Application layer module. On the other hand, if the packet received at the input port of Tftp module, it is transmitted to the output port of Tftp module.

22- Write C++ code for Mux_demux.cc as shown in Listing 9.19.

### Listing 9.19 Implementation of MuxDemux module

```cpp
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include "packet_m.h"
using namespace omnetpp;

class MuxDemuxModule:public cSimpleModule{
protected:
    Virtual void initialize() override;
    Virtual void handleMessage(cMessage *msg) override;
};

Define_Module(MuxDemuxModule);

void MuxDemuxModule::initialize()
{
    
void MuxDemuxModule::handleMessage(cMessage *msg)
{
    TcpIpPacket* pkt = check_and_cast<TcpIpPacket *>(msg);
    if(pkt->arrivedOn("muxDemuxGate$i",2)){
        if(pkt->getTcpHeader()==6){
            bubble("Packet Deliver through TCP");
            send(pkt,"muxDemuxGate$o",0);
        }
    }
    if(pkt->getTcpHeader()==13){
        bubble("Packet Deliver through UDP");
        send(pkt,"muxDemuxGate$o",1);
    }
}
)`
At handling phase (lines 13—28 of Listing 9.19), upon packet receiving at the input port of MuxDemux module (Line 25 of Listing 9.19), the packet will be sent to the output port (line 26 of Listing 9.19). On the other hand, if the packet is received at the input port of MuxDemux module at gate 2 (line 15 of Listing 9.19), TcpHeader value is verified to identify whether it requires TCP protocol (line 16 of Listing 9.19) or UDP protocol (line 20 of Listing 9.19).

23-

Write C++ code for TransportLayer.cc as shown in Listing 9.20.

```
#include<stdio.h>
#include<string.h>
#include<omnetpp.h>
#include<packet_m.h>
using namespace omnetpp;

class TransportLayer : public cSimpleModule{
  protected:
  virtual void initialize() override;
  virtual void handleMessage(cMessage *msg) override;
};

Define_Module(TransportLayer);

void TransportLayer::initialize(){
}

void TransportLayer::handleMessage(cMessage *msg) {
  TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
  if(pkt->arrivedOn("transGate$i",0)){
    send(pkt, "transGate$o",1);
  }
  else if(pkt->arrivedOn("phyGate$i",1))
    send(pkt, "transGate$o",0);
}
```

At handling phase (lines 13—20 of Listing 9.20), the Transport layer checks whether it receives a packet from the Application layer or the Network layer. Upon receiving a packet from the Network layer, the packet is forwarded to the Application layer and if it receives a packet from the Application layer, the packet is forwarded to the Network layer.

24-

Write C++ code for TCP.cc as shown in Listing 9.21.

```
#include<stdio.h>
#include<string.h>
#include<omnetpp.h>
#include<packet_m.h>
#include<vector>
using namespace omnetpp;
```
At handling phase (line 14—34 of Listing 9.21), upon packet reception at input port of Tcp module (line 17 of Listing 9.21), TcpHeader is assigned and attached to the packet (line 18 of Listing 9.21) and packet is sent to the MuxDemux module (line 20 of Listing 9.21). One the other hand, if the packet is received on the input port of Tcp module (line 22 of Listing 9.21), port number will be verified (lines 23 and 28 of Listing 9.21) and the packet is sent to the output gate 0 (line 26 of Listing 9.21).

25- Write C++ code for `UDP.cc` as shown in Listing 9.22.
Line 12: `Define_Module(Udp);`

Line 13: `void Udp::initialize(){  }`

Line 14: `void Udp::handleMessage(cMessage *msg){`
Line 15: `int transH =13;`
Line 16: `TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);`  
/* Condition is applied when message received on input port of UDP connected to Transport layer */
Line 17: `if(pkt->arrivedOn("udpGate$i",0)){`  
/*Addion of UDP header */
Line 18: `pkt->setTcpHeader(transH);`  
/* To send packet on output port of UDP connected to Transport layer */
Line 19: `bubble(" UDP header added ");`
Line 20: `send(pkt, "udpGate$o",1);`  
/* Condition is applied when message received on input port of UDP connected to Mux_Demux module */
Line 22: `if(pkt->arrivedOn("udpGate$i",1)){`  
/*Check Port No for TFTP */
Line 23: `if(pkt->getAppHeader().portNo == 100){`  
/*Remove TCP Header */
Line 24: `pkt->setTcpHeader(0);`  
Line 25: `bubble(" Read Port 100 Send to TFTP");`  
Line 26: `send(pkt, "udpGate$o",0);`  
Line 27: `}`
Line 28: `}`
Line 29: `}`

At handling phase (line 14—34 of Listing 9.22), upon packet reception at input port of Udp module (line 17 of Listing 9.22), transport layer header is assigned and attached to the packet (line 18 of Listing 9.22) and packet is sent to the Mux_Demux module (line 20 of Listing 9.22). On the other hand, if the packet is received on the input port of Udp module (line 22 of Listing 9.21), port number will be verified (lines 23 of Listing 9.22) and the packet is sent to the output gate 0 (line 26 of Listing 9.22).

26- Write C++ code for `NetworkLayer.cc` as shown in Listing 9.23.

### Listing 9.23 Implementation of NetworkLayer module

```cpp
Line 1: #include<stdio.h>
Line 2: #include<string.h>
Line 3: #include<omnetpp.h>
Line 4: #include<packet_m.h>
Line 5: #include<vector>
Line 6: using namespace omnetpp;

Line 7: class NetworkLayer : public cSimpleModule{
Line 8:   protected:
Line 9:     virtual void initialize() override ;
Line 10:     virtual void handleMessage(cMessage *msg) override ;
Line 11:   };

Line 12: Define_Module(NetworkLayer);
```
void NetworkLayer::initialize()
{
}

void NetworkLayer::handleMessage(cMessage *msg)
{
    const char *ipH = "IP Header Address 192.26.7.40";
    TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
    pkt->setIpHeader(ipH);
    if(pkt->arrivedOn("ipGate$i",0))
    {
        send(pkt, "ipGate$o",1);
    }
    else
    {
        if(pkt->arrivedOn("ipGate$i",1))
        {
            send(pkt, "ipGate$o",0);
        }
    }
}

At handling phase (lines 14—23 of Listing 9.23), the Network layer checks whether it receives a packet from the Data Link layer or the Transport layer. Upon receiving a packet from the Data Link layer, the packet is sent to the Transport layer. If the packet is received from the Transport layer, it adds IP header and forwards it to the Data Link layer.

27- Write following C++ code for DataLinkLayer.cc.

### Listing 9.24

```cpp
#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include <packet_m.h>
using namespace omnetpp;

class DataLinkLayer : public cSimpleModule{
    protected:
    virtual void initialize() override;
    Virtual void handleMessage(cMessage *msg) override;
};

Define_Module(DataLinkLayer);

void DataLinkLayer::initialize()
{
}

void DataLinkLayer::handleMessage(cMessage *msg)
{
    const char *dllH = "DLL Header MAC Address EC-F4-BB-57-51";
    TcpIpPacket *pkt = check_and_cast<TcpIpPacket *>(msg);
    pkt->setDllHeader(dllH);
    if(pkt->arrivedOn("dllGate$i",0))
    {
        send(pkt, "dllGate$o",1);
    }
    else
    {
        if(pkt->arrivedOn("dllGate$i",1))
        {
            send(pkt, "dllGate$o",0);
        }
    }
}
```

At handling phase (lines 13—22 of Listing 9.24), the DataLink layer checks whether it receives a packet from the Physical layer or the Network layer. Upon receiving a packet from Physical layer, the packet is sent to the Network layer. If the packet is received from the Network layer, it adds Data Link layer header and forwards it to the Physical layer.

28- Write C++ code for PhysicalLayer.cc as shown in Listing 9.25.
**Listing 9.25**

**Implementation of PhysicalLayer module**

Line 1: `#include<stdio.h>`
Line 2: `#include<string.h>`
Line 3: `#include<omnetpp.h>`
Line 4: `#include<packet_m.h>`
Line 5: `using namespace omnetpp;`

Line 6: `class PhysicalLayer : public cSimpleModule{`
Line 7: `protected:`
Line 8: `virtual void initialize() override;`
Line 9: `virtual void handleMessage(cMessage *msg) override;`
Line 10: `};`

Line 11: `Define_Module(PhysicalLayer);`

Line 12: `void PhysicalLayer::initialize(){   }`

Line 13: `void PhysicalLayer::handleMessage(cMessage *msg){`
Line 14: `TcpIpPacket *pkt = check_and_cast<TcpIpPacket*>(msg);`
Line 15: `if(pkt->arrivedOn("phyGate$i",0)){`
Line 16: `send(pkt, "phyGate$o",1);`
Line 17: `}`
Line 18: `else if(pkt->arrivedOn("phyGate$i",1))`
Line 19: `send(pkt, "phyGate$o",0);`
Line 20: `}`

At handling phase (line 13—20 of Listing 9.25), the Physical layer checks whether it receives a packet from the Data Link layer or Physical layer of connected PC. Upon receiving a packet from a Physical layer of connected PC, the packet is forwarded to the Data Link layer. If the packet is received from the Data Link layer, the packet is forwarded to the connected PC.

29- To create a network description file (for Point-to-Point network having two compound modules PC1 and PC2), follow step 31 of section 5.4.

30- Write code in the **Pt2Pt_Network.ned** file as shown in Listing 9.26.

**Listing 9.26**

**Implementation of Point-to-Point network**

Line 1: `package tcpipstack1.NEDSrc;`
Line 2: `// Creation of a Network consisting of two modules of type PC`
Line 3: `network Pt2Pt_Network{`
Line 4: `submodules:
Line 5: `PC1: PC;`
Line 6: `PC2: PC;`
Line 7: `connections:
Line 8: `PC1.gate++ <-> PC2.gate++;`}

31- To create omnetpp.ini file, follow steps 11—15 of section 8.3. Write Following code in omnetpp.ini file.

**Listing 9.27**

**Attributes of omnetpp.ini**

Line 1: `[General]`
Line 2: `network = tcpipstack3.NEDSrc.Pt2Pt_Network`
Whenever a simulation starts in OMNeT++, it reads NED and configuration file (omnetpp.ini) to assign parameter values. NED files are in first preference but if NED files contain no fixed values for parameters, then configuration file option is used to assign parameter values. Here, in Listing 9.27, Application IDs are provided in the configuration file.

32- Right-click on the project folder in Project Explorer window and Build Project.

33- Message Passing in the running simulation shown in Figure 9.3.

Three modules of Application layer (tftp, http, ftp) for PC1 and PC2 are shown in the left panel of Figure 9.3. In the right panel, multiplexing and demultiplexing process of tcp and udp modules of the Transport layer are shown.

**Practice Task**

1. Create a network having three nodes as shown below. Both PC1 and PC2 nodes are required to send Data packets to each other via intermediate node R. It is required to simulate the working of multiplexing/demultiplexing with five protocols (i.e., HTTP, FTP, TFTP, SNMP, and SMTP) at the Application layer.
GUI Interfaces and Enhancements

Learning Objectives:

After studying this chapter, you will be able to describe and use:

- Command-Line Interface
- Tekenv GUI-based Interface
- Qtenv GUI-based Interface
- Editing options of NED in Graphical Mode
- The availability of Graphical/Visualization options in implementation files
  - Use of Display String
  - Use of setTagArg()
  - Use of parse()
  - Use of bubble()
  - Use of WATCH Macro
  - Apply packet kind settings (to color packet)
10.1 OMNeT++ Execution Environment

Table 10.1 shows the summarized overview of simulation components in OMNeT++.

Table 10.1: Simulation components in OMNeT++

<table>
<thead>
<tr>
<th>Network Model:</th>
<th>simple/compound modules are used to build OMNeT++ based network simulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Structure:</td>
<td>is defined using NED language</td>
</tr>
<tr>
<td>Implementation Module:</td>
<td>for implementation, simulation kernel and C++ class library are used.</td>
</tr>
<tr>
<td>Configuration Module:</td>
<td>Within configuration module, the .ini files are used to assign parameter values to OMNeT++ model</td>
</tr>
<tr>
<td>Simulation Execution:</td>
<td>command-line (cmdenv) and graphical (Tcl/Tk-based and Qt-based) user interfaces are used for batch and interactive execution</td>
</tr>
<tr>
<td>Simulation Results:</td>
<td>can be seen as output vectors/scalars (text-based) files</td>
</tr>
<tr>
<td>HTML Documentation:</td>
<td>is generated from NED, MSG, and C++ files</td>
</tr>
</tbody>
</table>

For simulation execution, OMNeT++ provides two types of user interfaces i.e., command-line (Cmdenv) and graphical (Tcl/Tk-based (Tkenv) and Qt-based (Qtenv)) user interfaces.

10.2 Command-line Interface (Cmdenv)

Cmdenv is a small and portable user interface of OMNeT++. It is typically designed to perform batch executions. The Cmdenv compiles and executes in two modes (i.e., normal mode and express mode) on all (i.e., Windows, Unix, Linux etc.) platforms. Normal mode is used for debugging purposes and the express mode is used for long simulation executions where only periodical updates are required. The Cmdenv interface is capable of executing simulation runs already described in the configuration file. Through Cmdenv (with a comma-separated list of specific simulation run numbers), it is possible to specify the number of simulation runs to be executed. During a simulation run, Cmdenv shows the performance of the simulation execution (events/sec) with event sequence numbers and simulation/elapsed time. (An example simulation execution using the Cmdenv interface is shown in Figure 10.1. In addition, the output of Cmdenv is customizable through omnetpp.ini and log messages are written to the standard output file (called vector file).

Figure 10.1: Simulation execution using Cmdenv interface
10.3 Tkenv (Tcl/Tk-based) Interface

In OMNeT++, Tkenv is a portable graphical runtime interface, which is available to facilitate interactive simulation execution. Tkenv provides details related to the state of simulation at any instant of time during simulation execution. The most significant features of the Tkenv includes the message flow depiction (using animation), message flow log, event log, user-controlled run modes (i.e., step-wise, normal, fast, express etc.), simulation snapshots, network visualization, inspectors (to view contents of object/variables), and statistical visualization of simulation execution (as shown in Figure 10.2).

![Figure 10.2: Simulation execution using Tkenv interface](image)

10.4 Qtenv Interface

OMNeT++ provides another graphical runtime interface to facilitate users in terms of interactive simulation executions. Moreover, Qtenv can be used for 3D visualization of executing simulation as shown in Figure 10.3. Currently, the Qtenv interface is in the preliminary stage (considering OMNeT++ version 5.0).

![Figure 10.3: Simulation run in Qtenv](image)
10.5 Editing Options of NED in Graphical Mode

In the NED files’ graphical mode, simple module(s) are represented by icons. However, compound module(s) and network are represented by rectangles. The simple module can be placed within a compound module or networks as a submodule. Similarly, depending upon the nature of the connections, submodules connections can also be placed using Palette’s Connection Tool option as shown in Figure 10.4. In addition, source-code documentation can be viewed by placing mouse-pointer on any submodule or channel (as shown in Figure 10.5).

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>Icon specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple Host{</td>
<td></td>
</tr>
<tr>
<td>Line 2:</td>
<td>parameters:</td>
</tr>
<tr>
<td></td>
<td>// For module’s icon setting</td>
</tr>
<tr>
<td></td>
<td>// Use tag i as @display properties parameters</td>
</tr>
<tr>
<td>Line 3:</td>
<td>@display(&quot;i=device/pc2&quot;);</td>
</tr>
</tbody>
</table>

10.6 Graphical/Visualization Enhancements

In OMNeT++, GUI-based interfaces (such as Tkenv and Qtenv) offer features of the interactive simulation-based environment, visualization, and animations. To enhance visualization, various options (i.e., display string, bubbles, watches etc.), are available in both network description (.ned file) and implementation (.cc file) levels.

10.6.1 Use of Display Strings

Display strings are specified in NED files and can be manipulated through implementation files at runtime. Display string illustrates that how a module or channel can be visualized in GUI. In NED files, display strings (using @display properties) are used with submodules, compound modules, networks, connections, and messages. Each property has a single string value. Listing 10.1 to Listing 10.13 illustrates the use of @display string within NED code samples with the respective resultant output.
Table 10.2: Icon size

```plaintext
simple Host {
  parameters:
  // Use of "is" (icon size) tag in @display properties to set icon size
  @display("i=device/pc2;is=vl");
  gates:
  input in @directIn;
}
```

Values of is tag (line 3 of Listing 10.2) can be set as vs (for very small), s (for small), n (for normal), l (for large) and vl (for very large). The size’s effect can be seen at run time.

Table 10.3: Icon colorization

```plaintext
simple Host {
  parameters:
  // write name of the color as second parameter for tag i
  @display("i=device/pc2,yellow;is=l");
  gates:
  input in @directIn;
}
```

Change in icon color may be used to show the status of communication. Alternatively, line 3 of Listing 10.3 can be replaced with @display("i=device/pc2,yellow, 40;is=l"), where 40 represents the percentage in color change.

Table 10.4: Status icon

```plaintext
simple Host {
  parameters:
  // i2 tag can be used to set status icon
  @display("i=device/pc2,yellow;is=l;i2=status/noentry");
  gates:
  input in @directIn;
}
```

To show the status of a module during communication, a supporting short icon can be used that appears at the top-right corner of the main icon (for example host icon).
Listing 10.5  Icon positioning

Line 1: `simple` Host{
Line 2:   `parameters`:
Line 3:     `@display`("i=device/pc2");
Line 4:   `gates`:
Line 5:     `input` in `@directIn`;
Line 6: }
Line 7: `network` Example{
Line 8:   `submodules`:
Line 9:     H1: Host {
Line 10:     `parameters`:
Line 11:       `@display`("i=;black;p=100,100");
Line 12:     }
Line 13:     H2: Host {
Line 14:     `parameters`:
Line 15:       `@display`("i=;white;p=50,50");
Line 16:     }
Line 17: }

The tag `p` is required to specify the placement of a node in a simulation area. In the absence of tag `p`, modules placement is carried out using default layout algorithm.

Listing 10.6  Transmission range

Line 1: `simple` Host{
Line 2:   `parameters`:
Line 3:     `@display`("i=device/pc2");
Line 4:   `gates`:
Line 5:     `input` in `@directIn`;
Line 6: }
Line 7: `network` Example{
Line 8:   `submodules`:
Line 9:     H1: Host {
Line 10:     `parameters`:
Line 11:       `@display`("i=device/cellphone,black;p=100,100;r=50");
Line 12:     }
Line 13:     H2: Host {
Line 14:     `parameters`:
Line 15:       `@display`("i=device/antennatower,white;p=50,50;r=50");
Line 16:     }
Line 17: }
To show wireless range as filled color circle, display string should be set to

```display"i=device/antennatower,white;p=50,50;r=50, red";```

To set the width of range border, the display string should be set to

```display"i=device/antennatower,white;p=50,50;r=50, red";```

### Listing 10.7 Icon short text

| Line 1: | simple Host{               |
| Line 2: | parameters:               |
| Line 3: | @display("i=device/pc2"); |
| Line 4: | gates:                   |
| Line 5: | input in @directIn;      |
| Line 6: | }                        |

| Line 7: | network Example{           |
| Line 8: | submodules:               |
| Line 9: | H1: Host {                |
| Line 10: | parameters:              |
| Line 11: | @display("i=,black;p=100,100"); |
| Line 12: | }                       |
| Line 13: | H2: Host {                |
| Line 14: | parameters:              |
| Line 15: | // Use of t (text) tag within @display to show text on icon |
| Line 16: | @display("i=,white;p=50,50;t=rcvd:93 sent:10"); |
| Line 17: | }                       |

### Listing 10.8 Icon tooltip

| Line 1: | simple Host{               |
| Line 2: | parameters:               |
| Line 3: | @display("i=device/pc2,yellow;is=l;i2=status/noentry"); |
| Line 4: | gates:                   |
| Line 5: | input in @directIn;      |
| Line 6: | }                        |

| Line 7: | network Example{           |
| Line 8: |                           |
| Line 9: |                           |
| Line 10: |                           |
| Line 11: |                           |
| Line 12: |                           |
| Line 13: |                           |
| Line 14: |                           |
| Line 15: |                           |
| Line 16: |                           |
| Line 17: |                           |
Line 8:   **submodules:**
Line 9:     H1:Host {
Line 10:       **parameters:**
Line 11:       // Use of `tt` (ToolTip) tag within `@display` to set Tooltip for module
Line 12:       `@display`("i=,black;p=100,100;tt= My Host1");
Line 13:     }

Use **tt** tag to display tooltip information of a node in the network.

**Listing 10.9 Arrangement of submodules' vector elements**

Line 1:     **simple** Host{
Line 2:       **parameters:**
Line 3:       `@display`("i=device/pocketpc,white");
Line 4:       **gates:**
Line 5:     } **input** in `@directIn`

Line 7: **network** Example{
Line 8:     **submodules:**
Line 9:     H[20]:Host {
Line 10:       **parameters:**
Line 11:       // Use `m` tag in `@display` to arrange submodule vector elements in matrix
Line 12:       `@display`("p=,,m,4,50,50");
Line 13:     }

Line 14: }

```
```
P-tag can be used to arrange submodules' vector elements in the form of a matrix (as shown above).

**Listing 10.10** Arrangement of submodules' vector elements

<table>
<thead>
<tr>
<th>Line 1:</th>
<th><code>simple Host1{</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2:</td>
<td><code>parameters:</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>@display(&quot;i=device/pc2,white&quot;);</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>gates:</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>input in @directIn;</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

| Line 7: | `network Example1{`                                   |
| Line 8: | `submodules:`                                         |
| Line 9: | `H[20]:Host1 {`                                       |
| Line 10:| `parameters:`                                        |
| Line 11:| `// Use ri tag in @display to arrange submodule vector elements in ring` |
| Line 12:| `@display("p=,,ri,50,50");`                         |
| Line 13:| `}`                                                   |

P tag can be used to arrange submodules' vector elements in the form of a ring (as shown above).

**Listing 10.11** Arrangement of submodules' vector elements

<table>
<thead>
<tr>
<th>Line 1:</th>
<th><code>simple Host1{</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2:</td>
<td><code>parameters:</code></td>
</tr>
<tr>
<td>Line 3:</td>
<td><code>@display(&quot;i=device/pc2,white&quot;);</code></td>
</tr>
<tr>
<td>Line 4:</td>
<td><code>gates:</code></td>
</tr>
<tr>
<td>Line 5:</td>
<td><code>input in @directIn;</code></td>
</tr>
<tr>
<td>Line 6:</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

| Line 7: | `network Example1{`                                   |
| Line 8: | `submodules:`                                         |
| Line 9: | `H[4]:Host1 {`                                       |
| Line 10:| `parameters:`                                        |
| Line 11:| `// Use r tag in @display to arrange submodule vector elements in a row` |
| Line 12:| `@display("p=100,100,r,50");`                       |
| Line 13:| `}`                                                   |
\(P\)-tag can be used to arrange submodules’ vector elements in the form of row. (Similarly, the \(P\)-tag can be employed for column topology using @display("p=100,100,c,50").)

**Listing 10.12** Channel/Connection display string

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>simple Host{</td>
</tr>
<tr>
<td>2</td>
<td>parameters:</td>
</tr>
<tr>
<td>3</td>
<td>@display(&quot;i=device/pc2,yellow;is=1;i2=status/noentry&quot;);</td>
</tr>
<tr>
<td>4</td>
<td>gates:</td>
</tr>
<tr>
<td>5</td>
<td>input in;</td>
</tr>
<tr>
<td>6</td>
<td>output out;</td>
</tr>
<tr>
<td>7</td>
<td>}</td>
</tr>
<tr>
<td>8</td>
<td>network Example{</td>
</tr>
<tr>
<td>9</td>
<td>submodules:</td>
</tr>
<tr>
<td>10</td>
<td>H1: Host {</td>
</tr>
<tr>
<td>11</td>
<td>parameters:</td>
</tr>
<tr>
<td>12</td>
<td>@display(&quot;i=,black;p=100,100;tt= My Host1&quot;);</td>
</tr>
<tr>
<td>13</td>
<td>}</td>
</tr>
<tr>
<td>14</td>
<td>H2: Host {</td>
</tr>
<tr>
<td>15</td>
<td>parameters:</td>
</tr>
<tr>
<td>16</td>
<td>@display(&quot;i=,white;p=500,250&quot;);</td>
</tr>
<tr>
<td>17</td>
<td>}</td>
</tr>
<tr>
<td>18</td>
<td>connections:</td>
</tr>
<tr>
<td>19</td>
<td>// Use t tag in @display for text</td>
</tr>
<tr>
<td>20</td>
<td>// Use t attribute for t tag to display text at top</td>
</tr>
<tr>
<td>21</td>
<td>// Alternative to t attribute, l (left) and r (right) can be used</td>
</tr>
<tr>
<td>22</td>
<td>// d attribute of ls stands for dotted line</td>
</tr>
<tr>
<td>23</td>
<td>H1.out --&gt; { @display(&quot;t=10Gbps,t,black;ls=red,4,d&quot;); } --&gt; H2.in;</td>
</tr>
<tr>
<td>24</td>
<td>H1.in &lt;-- { @display(&quot;ls=red,4,d;tt=Cable&quot;); } &lt;-- H2.out;</td>
</tr>
</tbody>
</table>

Connection display can be supported by various tags i.e., \(tt\) for the tooltip, \(ls\) for line style and color, \(m\) for orientation and positioning, \(t\) for text etc. In addition, similar to submodules, connections may inherit display string from channel types.
Listing 10.13  

```
1:  simple Host {
2:     parameters:
3:       @display("i=device/pc2,yellow");
4:     gates:
5:       input in;
6:     output out;
7: }

8:  network Example{
9:    // For background visualization
10:   // bgb = 900, 400 means simulation area 900, 400
11:   // yellow parameter value for background color
12:   // black parameter value stands for border color
13:   // 5 value is for border size
14:   // bgtt means tooltip is World net
15:   // bgi stands for background image
16:   // bgg = 100 means major axis at each 100 space
17:   // 2 means major box divided into 2 sub-divisions
18:   // black means both major and minor axis color is black
19:   // bgu means background simulation area units
20:    @display("bgb=900,400,yellow,black,5;bgtt=World net;
21:       bgi=maps/world;bgg=100,2,black;bgu=km");
22: }

23: submodules:
24:  H1: Host {
25:     parameters:
26:       @display("i=,black;p=150,150;tt= My Host1");
27:  }
28:  H2: Host {
29:     parameters:
30:       @display("i=,white;p=600,150");
31:  }
32: connections:
33:  H1.out --> { @display("t=10Gbps,t,black;ls=red,4,d"); } --> H2.in;
34:  H1.in <-- { @display("ls=red,4,d;tt=Cable"); } <-- H2.out;
```
simulation area), bgg (for background color and spacing), bgu (for measurement units), bgtt (for background tooltip) etc. The use of these tags has been explained with the help of following example.

### 10.6.2 Use of setTagArg() - Change in Display Strings at Runtime

During the simulation execution, occasionally it is helpful to manipulate module’s icon, color, position etc. Module’s manipulation at runtime can be done through objects of cDisplayString class. The cDisplayString class assists in finding out tags of display string associated with channels, modules, and gates etc. The getDisplayString() method is required to obtain the pointer of cDisplayString object. Three methods (setTagArg, parse, and removeTag() with cDisplayString object can be used to overwrite display strings. Listings 10.14—10.17 illustrate the manipulation of display strings at runtime.

Listings 10.14 and 10.15 show icon changes at run time.

#### Listing 10.14  Network description for Listing 10.15

```plaintext
// Network Description for listing 10.15
Line 1: simple Host{
Line 2: parameters:
Line 3: @display("i=device/pc2,yellow");
Line 4: gates:
Line 5: input in;
Line 6: output out;
Line 7: }

Line 8: network Example{
Line 9: submodules:
Line 10: H1: Host {
Line 11: parameters:
Line 12: @display("i=,yellow;p=150,150;tt=My Host1;i2=status/off");
Line 13: }
Line 14: H2: Host {
Line 15: parameters:
Line 16: @display("i=,yellow;p=600,150;i2=status/off");
Line 17: }
Line 18: connections:
Line 19: H1.out --> { } --> H2.in;
Line 20: H1.in <-- { } <-- H2.out;
Line 21: }
```

#### Listing 10.15  Use of setTagArg() to change icon at run-time

```plaintext
// implementation of listing 10.14 shows the use of setTagArg
Line 1: #include<omnetpp.h>
Line 2: #include<string.h>
Line 3: #include<stdio.h>

Line 4: using namespace omnetpp;
Line 5: class Host:public cSimpleModule{
Line 6: protected:
Line 7: virtual void initialize()override ;
// Declare method to set simulation area (simulation background)
Line 8: virtual void handleMessage(cMessage *msg) override;
Line 9: }
```

128
Line 10: Define_Module(Host);

Line 11: void Host::initialize()

    /*
    Calling of updateDisplay method
    getSystemModule() returns top-level module of current simulation
    Here, top-level module is Example network
    */
    Line 12:     cMessage *msg = new cMessage("Pkt");
    Line 13:     if(strcmp("H1", getName()) == 0)
    Line 14:         send(msg,"out");
    Line 15: }

Line 16: void Host::handleMessage(cMessage *msg){

    /*
    hasGUI() returns true for GUI environment (in case of Tkenv/Qtenv) and
    returns false in Cmdenv. Upon returing values, modules decide about the
    bothering of visualization
    */
    Line 17: if(hasGUI()){

        /*
        getDisplayString return displayString object which ultimately defines
        presentation. setTagArg() is used to update text
        */
        Line 18:          getDisplayString().setTagArg("i2",0,"status/red");
        Line 19:          getDisplayString().setTagArg("i",1,"blue");
        Line 20:      }
        Line 21:      send(msg,"out");
        Line 22: if(hasGUI()){
        Line 23:          getDisplayString().setTagArg("i2",0,"status/off");
        Line 24:          getDisplayString().setTagArg("i",1,"yellow");
        Line 25:      }
        Line 26:    }

Listing 10.16 and Listing 10.17 elaborate the change in simulation (background) area at run time.
// Network Description for listing 10.15
Line 22:    simple Host{
Line 23:        parameters:
Line 24:            @display("i=device/pc2,yellow");
Line 25:        gates:
Line 26:            input in;
Line 27:            output out;
Line 28: }

Line 29:    network Example{
Line 30:        submodules:
Line 31:            H1: Host {
Line 32:                parameters:
Line 33:                    @display("i=,yellow;p=150,150;tt=My Host1;i2=status/off");
Line 34:                }
Line 35:            H2: Host {
Line 36:                parameters:
Line 37:                    @display("i=,yellow;p=600,150;i2=status/off");
Line 38:                }
Line 39:        connections:
Line 40:            H1.out --> { } --> H2.in;
Line 41:            H1.in <-- { } <-- H2.out;
Line 42: }

// implementation of listing 10.14 shows the use of setTagArg
Line 1:    #include<omnetpp.h>
Line 2:    #include<string.h>
Line 3:    #include<stdio.h>

Line 4:    using namespace omnetpp;
Line 5:    class Host:public cSimpleModule{
Line 6:        protected:
Line 7:            virtual void initialize() override ;
// Declare method to set simulation area (simulation background)
Line 8:            virtual void updateDisplay(cModule *simPtr) override ;
Line 9:            virtual void handleMessage(cMessage *msg) override ;
Line 10:};

Line 11: Define_Module(Host);

Line 12: void Host::initialize(){
/*
Calling of updateDisplay method
getSystemModule() returns top-level module of current simulation
Here, top-level module is Example network
*/
Line 13:    updateDisplay(getSystemModule());
Line 14:    cMessage *msg = new cMessage("Pkt");
Line 15:    if(strcmp("H1", getName()) == 0)
Line 16:        send(msg,"out");
Line 17:}
Line 18: `void Host::handleMessage(cMessage *msg){`

`/*
hasGUI() returns true for GUI environment (in case of Tkenv/Qtenv) and returns false in Cmdenv. Upon returning values, modules decide about the bothering of visualization */
*/`

Line 19: `if(hasGUI()){`

`/*
getDisplayString return displayString object which ultimately defines presentation. setTagArg() is used to update text */`

Line 20: `getDisplayString().setTagArg("i2",0,"status/red");`

Line 21: `getDisplayString().setTagArg("i",1,"blue");`

Line 22: `}`

Line 23: `send(msg,"out");`

Line 24: `if(hasGUI()){`

Line 25: `getDisplayString().setTagArg("i2",0,"status/off");`

Line 26: `getDisplayString().setTagArg("i",1,"yellow");`

Line 27: `}`

Line 28: `}`

Line 29: `void Host::updateDisplay(cModule *simPtr){`

Line 30: `cDisplayString& ds = simPtr->getDisplayString();`

`// "bgb" is also related to module background
// "bgb" with 0 means width of the background module (simulation area)`

Line 31: `ds.setTagArg("bgb",0,700);`

`// "bgb" with 1 means height of the background module (simulation area)`

Line 32: `ds.setTagArg("bgb",1,400);`

`// bgg is used for grid tick distance
// "bgg" with 0 means Distance b/w two major ticks measured in units`

Line 33: `ds.setTagArg("bgg",0,100);`

`// "bgg" with 1 means minor ticks per major ticks`

Line 34: `ds.setTagArg("bgg",1,3);`

`// "bgg" with 2 means color of grid lines`

Line 35: `ds.setTagArg("bgg",2,"gray");`

`// bgu is related to background distance units
// "bgu" with 1 means the name of the distance unit`

Line 36: `ds.setTagArg("bgu",0,"km");`

Line 37: `}`
10.6.3 Use of parse()

Alternatively, parse() method can be used to set display properties at runtime. For example:

Code snippet for setting module’s position

```c
    cDisplayString& dsObj = getDisplayString();
    dsObj.parse("p=40,20");
```

Code snippet for setting icon and status icon

```c
    cDisplayString& dsObj = getDisplayString();
    dsObj.parse("i=device/pc;i2=status/off");
```

Code snippet for setting background with grid lines

```c
    cDisplayString& pObj = getParentModule()->getDisplayString();
    pObj.parse("bgi=maps/asia;bgg=50,2");
```

Code snippet for setting channel color

```c
    cDisplayString& ccObj = gate("out")->getDisplayString();
    ccObj.parse("ls=red");
```

10.6.4 Use of Bubbles

In OMNeT++ based simulations, bubble() method is employed to show module’s short momentary message (or notification). The `bubble()` method takes as an input parameter a string (const char* pointer). Listing 10.18 shows the use of `bubble(string)` method. Use network description (.ned file) of Listing 10.14 for this network setup.

**Listing 10.18** Use of bubble with String parameter

- Line 1: `#include<omnetpp.h>`
- Line 2: `#include<string.h>`
- Line 3: `#include<stdio.h>`
using namespace omnetpp;

class Host:public cSimpleModule{
    protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(Host);

void Host::initialize(){
    cMessage *msg = new cMessage("Pkt");
    if(strcmp("H1", getName()) == 0)
        send(msg, "out");
}

void Host::handleMessage(cMessage *msg){
    if(hasGUI()){
        getDisplayString().setTagArg("i2",0,"status/red");
        getDisplayString().setTagArg("i",1,"blue");
        // To display bubble on message reception
        bubble("Message Received");
    }
    send(msg, "out");
    if(hasGUI()){
        getDisplayString().setTagArg("i2",0,"status/off");
        getDisplayString().setTagArg("i",1,"yellow");
    }
}

Listing 10.19 shows the use of bubble(char array) method. Use network description (.ned file) of Listing 10.14 for this network setup.

---

Listing 10.19  Use of bubble with char * pointer

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

using namespace omnetpp;

class Host:public cSimpleModule{
    private:
    int seqNo;
    protected:
virtual void initialize() override;
virtual void handleMessage(cMessage *msg) override;

Define_Module(Host);

void Host::initialize()
{
  seqNo = 0;
cMessage *msg = new cMessage("Pkt");
  if(strcmp("H1", getName()) == 0)
    send(msg,"out");
}

void Host::handleMessage(cMessage *msg){
  if(hasGUI()){
    getDisplayString().setTagArg("i2",0,"status/red");
    getDisplayString().setTagArg("i",1,"blue");
    // To display bubble on message reception with received packet number
    char msgText[32];
    sprintf(msgText,"Received Pkt-%d",++seqNo);
    bubble(msgText);
  }
}

10.6.5 Use of WATCH macro and Icon Short Text changes in Dynamic Scenario

In OMNeT++ simulations, icon short text can be employed at the time of simulation execution. In OMNeT++, basic WATCH() macro facilitates programmer to inspect variable values using Tkenv/Qtenv interfaces. The STL WATCH macros are available to describe standard C++ container classes (of Standard Template Library (STL)). For user-defined classes, structured WATCH macros are also available.

Listing 10.20 Use of WATCH() and icon short text in dynamic scenario

#include <stdio.h>
#include <string.h>
#include <omnetpp.h>
#include "omnetpp.h"
using namespace omnetpp;
class Host : public cSimpleModule{
private:
long numSent;
long numReceived;

protected:
virtual void refreshDisplay() const override;
virtual void initialize() override;
virtual void handleMessage(cMessage *msg) override;

};

void Host::initialize()
{
// Initialize variables
numSent = 0;
numReceived = 0;

// Module 0 sends the first message
if (strcmp("H1", getName()) == 0) {
    cMessage *start = new cMessage("Pkt");
scheduleAt(0.0, start);
}

void Host::handleMessage(cMessage *msg){
if (strcmp("H2", getName()) == 0) {
    numReceived++;
    cMessage *ack = new cMessage("Ack");
send(ack,"out");
}
if (strcmp("H1", getName()) == 0) {
    numSent++;
    cMessage *msg = new cMessage("Pkt");
send(msg,"out");
}

/* Dedicated method in OMNeT++ 5.0 for visualization code. This method is
invoked under GUI mode right before display updates i.e.,
 - after setting up network
 - after every event in step/run mode
 - after every batch of events in fast/express mode
 - after finalization
*/
void Host::refreshDisplay() const{
if (strcmp("H1", getName()) == 0) {
    char buf[40];
sprintf(buf, "sent: %ld", numSent);
    getDisplayString().setTagArg("t", 0, buf);
}
if (strcmp("H2", getName()) == 0) {
    char buf[40];
sprintf(buf, "rcvd: %ld", numReceived);
    getDisplayString().setTagArg("t", 0, buf);
}
Variables can be watched in each node’s detail view, if \texttt{WATCH}() macro is used as shown in Figures below.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Figure 1: Network Description for Listing 10.22}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Figure 2: Packet color setting at run-time}
\end{figure}

### 10.6.6 Packet Kind Setting

In OMNeT++ simulations, different colors can be used to represent different packets to enhance the user visualization. Listing 10.21 and Listing 10.22 explain color setting related code for different packets at runtime.

#### Listing 10.21

// Network Description for Listing 10.22  

```c
// Network Description for listing 10.22
Line 1:   simple Node{
Line 2:     parameters:
Line 3:       @display("i=block/routing");
Line 4:     gates:
Line 5:       input in;
Line 6:       output out;
Line 7:   }
Line 8:   network MyNetwork{
Line 9:     submodules:
Line 10:       N1: Node {
Line 11:         parameters:
Line 12:           @display("i=,cyan");
Line 13:       }
Line 14:       N2: Node {
Line 15:         parameters:
Line 16:           @display("i=,gold");
Line 17:       }
Line 18:     connections:
Line 19:       N1.out --> { delay = 100ms; } --> N2.in;
Line 20:       N1.in <-- { delay = 100ms; } <-- N2.out;
Line 21:   }
```

#### Listing 10.22

// Implementation of listing 10.21 shows the use of setKind(int)  

```c
// Implementation of listing 10.21 shows the use of setKind(int)
Line 1:   #include <string.h>
Line 2:   #include <omnetpp.h>
Line 3:   using namespace omnetpp;
Line 4:   class Node : public cSimpleModule{
Line 5:     protected:
Line 6:       virtual void initialize() override;
Line 7:       virtual void handleMessage(cMessage *msg) override;
Line 8:   };
Line 9:   Define_Module(Node);
```
Line 10: void Node::initialize()
Line 11: if (strcmp("N1", getName()) == 0) {
Line 12: cMessage *msg = new cMessage("RTS");
Line 13: send(msg,"out");
Line 14: }
Line 15: }

Line 16: void Node::handleMessage(cMessage *msg){
Line 17: if(strcmp("RTS",msg->getName())==0){
Line 18: msg = new cMessage("CTS");
// To color packet, use setKind(int) function
Line 19: msg->setKind(1);
Line 20: send(msg,"out");
Line 21: }else if(strcmp("CTS",msg->getName())==0){
Line 22: msg = new cMessage("DATA");
Line 23: msg->setKind(4);
Line 24: send(msg,"out");
Line 25: }else if(strcmp("DATA",msg->getName(),3)==0){
Line 26: msg = new cMessage("ACK");
Line 27: msg->setKind(3);
Line 28: send(msg,"out");
Line 29: }else if(strcmp("ACK",msg->getName())==0){
Line 30: msg = new cMessage("RTS");
Line 31: msg->setKind(3);
Line 32: send(msg,"out");
Line 33: }
Line 34: }

Practice Task
1. Create a network having three nodes as shown in Figure below. PC1 sends request (REQ) packet to R. R processes the REQ packet and sends back reply (REP) packet to PC1. On receiving REP, PC1 will send data packet (DPkt). Upon receiving DPkt, R will forward it to PC2 using the same REQ-REP mechanism. Similarly, transfer a packet from PC2 to PC1 using REQ-REP mechanism. You are required to use different colors for each packet type. Each node icon should change its color upon packet reception. Use overlay icon to show the current status of each node with color changing. The short text is required to be updated on end nodes at each packet reception. Use WATCH macro to observe the status of sent packets. Event progress is required to be shown through message bubbles.
Case Study: Client-Server Architecture

Learning Objectives:

After studying this chapter, you will be able to describe:

- the basics of Client-Server architecture
- the difference between activity() and handleMessage() functions of OMNeT++
- the implementation details of Client-Server architecture in OMNeT++
11.1 Client-Server Architecture

On the basis of architecture, computer networks can be classified as Client-Server and Peer-to-Peer networks. The client-server architecture is an example of distributed application structure consisting of resource/service provider (named as Server) and service requester (named as Client). Server and Client(s) may reside in same or different computer networks. The Client-Server architecture is also known as producer-consumer computer architecture where Server acts as service producer and Client acts as the on-demand service consumer. Services of a Server may include storage, computing power, file/printer sharing etc. On the other hand, clients are not responsible for sharing any kind of resources but to request available resources from the Server.

The Internet is the most common example of Client-Server architecture, where millions of clients request for website data available on various web servers all over the world. In this chapter, we have explained the working of Client-Server architecture already implemented in OMNeT++ (known as DYNA).

In OMNeT++, DYNA represents a Client-Server architecture where a server may connect to several clients through a switch as shown in Figure 11.1.

![Figure 11.1: DYNA – client/server architecture](image)

DYNA implementation consists of six module types i.e.,

- **Server Module**: represents simple server which is responsible for serving multiple connection requests from client
- **ServerProcess Module**: is responsible for handling one connection in the server
- **Client Module**: is responsible for establishing connection with server to exchange data
- **Switch Module**: is responsible for the reception/forwarding of packets from clients to the server and vice versa
- **ClientServer Module**: represents network consisting of Server, Switch, and various Clients
11.2 Working of DYNA
The number of clients in the DYNA example can be configurable and each client in the network periodically sends connection requests to the server. After connection establishment, the client sends a number of queries before connection closes. On the other hand, the Server is responsible for serving concurrent incoming connections through a separate process implemented as ServerProcess module (part of Server as can be seen in Object tree (Figure 11.2)).

11.3 Details of DYNA Implementation
DYNA Project in OMNeT++ consists of five network description files (i.e., Client.ned, Server.ned, Switch.ned, ServerProcess.ned, ClientServer.ned) with corresponding implementation files (Client.cc, Server.cc, Switch.cc, ServerProcess.cc). In addition, few configuration files and message files are also part of DYNA project. All DYNA project files are shown in Figure 11.3.

Figure 11.2: Object tree of server node Figure 11.3: Project files in project explorer

Implementation details available in these files have been properly explained through incorporated comments in Listing 11.1 to Listing 11.13.

Listing 11.1

# In .ini files, lines starting with `#' are comments
# General configurations are applicable for all DYNA scenarios
Line 1:  [General]
# network file name is ClientServer.ned
Line 2:  network = ClientServer
# Snapshot file is used to save enough details for debugging
# By default, all simulation modules written in snapshot-file
# Snapshot helps to trace changes in variable/objects values
Line 3:  snapshot-file = ClientServer.sna
Vector (.vec) and Scalar (.sca) files record simulation run
# Vector files record data values as a function of time
# Scalar files record aggregate values at the end of the simulation
# In DYNA, upon simulation completion, .vec file found in Result folder
# Clicking on a vector file first time the ide creates an .anf file
# .anf is used to store information about how to create a chart
# Chart may include data selection, preprocessing, chart properties
Line 4: output-vector-file = ClientServer.vec
# Simulation time limit is 50000 second
# It shows the time duration of simulation
Line 5: sim-time-limit = 50000s
# CPU time limit is set to 600 nsecond
# It shows the time that simulation takes to complete
Line 6: cpu-time-limit = 600s
# Total-stack specifies maximum memory for activity simple module stack
# MiB (Mega byte = 1.04858e6B) unit recognized as constants in OMNeT++
# By default stack size is 1 MiB, but
# Upon getting co-routine stack allocation error, increase stack size
Line 7: total-stack = 7MiB  # increase if necessary
# Exponential connection inter-arrival time describes continuous and
# independent time interval between connection requests
# Here, exponential distribution is mentioned with mean of 10 seconds
Line 8: ClientServer.client[*].connIaTime = exponential(10s)
# Exponential query inter-arrival time describes continuous and
# independent time interval between query requests
# Here, exponential distribution is mentioned with mean of 2 seconds
Line 9: ClientServer.client[*].queryIaTime = exponential(2s)
# For number of queries, normal distribution truncated to non-negative #
# values is mentioned with mean of 3 and standard deviation of 1
Line 10: ClientServer.client[*].numQuery = truncnormal(3,1)
# Request processing time on server is 0.2 seconds
Line 11: ClientServer.server.processingTime = 0.2s
# Configuration option with number of clients ask from user
Line 12: [Config User]
Line 13: description = "ask number of client computers"
Line 14: ClientServer.numClients = ask
# Configuration option for small network with 8 client computers
Line 15: [Config Small]
Line 16: description = "8 client computers"
Line 17: ClientServer.numClients = 8
# Configuration option for medium network with 20 client computers
Line 18: [Config Medium]
Line 19: description = "20 client computers"
Line 20: ClientServer.numClients = 20

Listing 11.2 shows that DynaDataPacket fundamentally consists of source address, destination address, server process ID, and payload. Packet types have been declared as an enum data type (i.e., connection req, connection ack, disconnection req, disconnection ack, and data).

Listing 11.2  Attributes of data packet

// Enum is used to define integral constants to a range of values
// Packet types in DYNA are:
// connection/disconnection request, acknowledgement and data
// Required packet types have been defined as enum constants
Line 1: enum DynaPacketType{
DYNA_CONN_REQ = 0;
DYNA_CONN_ACK = 1;
DYNA_DISC_REQ = 2;
DYNA_DISC_ACK = 3;
 DYNA_DATA = 4;
};

// DYNA packet have three filed
// Source/destination address and server process ID
fullname
DynaPacket{
int srcAddress;
int destAddress;
int serverProcId;
};

// DynaDataPacket inherits attributes of DynaPacket and adds payload
fullname
DynaDataPacket extends DynaPacket{
string payload;
};

Listing 1.3 describes the Server design in Network Description file.

Listing 1.3

Server.ned

// Server module consists of a pair of input/output gate named as port
// In addition, request processing time is declared as 0.2 seconds
// @unit property is used to associate measurement unit (S or Seconds)
// @prompt property is used to define a prompt string for parameter
// The prompt string is used for interactive prompts if required during
// simulation runtime
 fullname
simple Server{
parameters:
  double processingTime @unit(s)
    @prompt("Time to process a query") = default(0.2s);
@gates:
  inout port;
}
Listing 11.5 describes the topology of DYNA network.

Listing 11.5

```plaintext
// DYNA network topology consists of a single server connected to
// single/multiple clients through a switch

network ClientServer{
    parameters:
    // Number of clients in default is 4 but can be configurable
    int numClients = default(4);
    submodules:
    server: Server{
        parameters:
        // Here, @display used to specify position of server in simulation area
        @display("p=210,70");
    }
    switch: Switch{
        parameters:
        // pkRate for processing should be >= numClients,
        // If pkRate < numClients, switch will become the bottleneck
        pkRate = 1.5 * numClients;
        // queueMaxLen representing buffer size of max 20 packets
        queueMaxLen = 20;
        // Here, @display used to specify position of server in simulation area
        @display("p=210,170");
        gates:
        // Number of switch gate pairs depends upon number of clients
        port[numClients+1];
    }
    client[numClients]: Client{
        parameters:
        // Timeout value for connection/query request is mentioned as 5 seconds
        timeout = 5s;
        // p tag can be used to arrange elements
        // arrangement can be in row, column, matrix or ring form
        // Here, @display property is used to arrange elements in a matrix
        // For matrix, p should be p = xpos, ypos, m, noOfCols, deltaX, deltaY
        // xpos, ypos arguments specify starting element position
        // m stands for matrix arrangement
        // noOfCols for number of columns
        // deltaX, deltaY arguments for units between rows and columns
        @display("p=70,270,m,10,80");
    }
    for i=0..numClients-1 {
        client[i].port<-->{delay = 10ms;}<-- switch.port[i];
    }
}
```

// Number of switch gate pairs depends on server + number of clients
// for loop is utilized to connect switch module to each client module
Line 26:    server.port<--{delay = 10ms;}{--->switch.port[numClients];
Line 27:  }

Listing 11.6 describes the ServerProcess of DYNA network.

```
Listing 11.6  ServerProcess.ned

// Server Process module contains one gate of input type
Line 1:  simple ServerProcess{
Line 2:    gates:
Line 3:       input in;
Line 4:  }
```

Listing 11.7 describes the Switch module of DYNA network.

```
Listing 11.7  Switch.ned

Line 1:  simple Switch{
Line 2:    parameters:
Line 3:       double pkRate; // for packet arrival rate at switch
Line 4:       int queueMaxLen; // for maximum switch buffer size
Line 5: @display("i=block/switch;q=queue");
Line 6:    gates:
Line 7: // Single gates or gate vectors can be created in OMNeT++
Line 8: // Gate vector size can be given in the declaration i.e. [5]
Line 9: // Gate vector size can be left open as "[ ]", can be specified later
Line 10: // upon using the module as a submodule in a compound module
Line 11: // However, without specifying one can create connections using gate++
Line 12: // The gate++ operator automatically expands the gate vector
Line 13:    inout port[];
Line 14:  }
```

Listing 11.8 describes declaration of packet kind values as an enum.

```
Listing 11.8  dyna.h

Line 1:    #include <omnetpp.h>
Line 2:  // Message kind values (or packet types) declared as enum constants
Line 3:  enum{
Line 4:    CONN_REQ,
Line 5:    CONN_ACK,
Line 6:    DATA_QUERY,
Line 7:    DATA_RESULT,
Line 8:    DISC_REQ,
Line 9:    DISC_ACK
Line 10:  };
```

Listing 11.9 describes the implementation of the Server.ned file.

```
Listing 11.9  Server.cc

Line 1:    #include "DynaPacket_m.h"
Line 2: using namespace omnetpp;
Line 3: class Server : public cSimpleModule{
Line 4:   private:
Line 5:    // cModuleType is abstract class for creating module of a specific type
cModuleType *srvProcType;
protected:
virtual void initialize() override;
virtual void handleMessage(cMessage *msg) override;
};

Define_Module(Server);
void Server::initialize()
{
    srvProcType = cModuleType::find("ServerProcess");
}

void Server::handleMessage(cMessage *msg)
{
    DynaPacket *pk = check_and_cast<DynaPacket *>(msg);
    if(pk->getKind() == DYNA_CONN_REQ) {
        cModule *mod = srvProcType->createScheduleInit("serverproc", this);
        EV << "DYNA_CONN_REQ: Created process ID="
            << mod->getId() << endl;
        sendDirect(pk, mod, "in");
    } else{
        int serverProcId = pk->getServerProcId();
        EV << "Redirecting msg to process ID=
              " << serverProcId << endl;
        cModule *mod = getSimulation()->getModule(serverProcId);
        if(!mod) { //If mod is null pointer
            EV << " That process already exited, deleting msg\n";
            delete pk;
        } else { 
            sendDirect(pk, mod, "in");
        }
    }
}

Listing 11.10 describes the ServerProcess module of DYNA network and is related to the implementation of ServerProcess.ned. ServerProcess dynamically launches the process in the server.
Listing 11.10

ServerProcess.cc

Line 1: #include "DynaPacket_m.h"
Line 2: using namespace omnetpp;
Line 3: #define STACKSIZE  16384

Line 4: class ServerProcess : public cSimpleModule{
Line 5:   public:
Line 6:     ServerProcess():cSimpleModule(STACKSIZE) { }
Line 7:     virtual void activity() override;
Line 8:   };  
Line 9: Define_Module(ServerProcess);
Line 10: void ServerProcess::activity(){
    /*
    To retrieve parameters.
    Here, getParentModule() returns the module containing this module;
    It means it will return instance of ClientServer
    */
    Line 11:     cPar& processingTime =
        getParentModule()->par("processingTime");
    Line 12:     cGate *serverOutGate = getParentModule()->gate("port$o");
    Line 13:     int clientAddr = 0, ownAddr = 0;
    Line 14:     WATCH(clientAddr);
    Line 15:     WATCH(ownAddr);
    Line 16:     DynaPacket *pk;
    Line 17:     DynaDataPacket *datapk;
    // To receive the CONN_REQ
    Line 18:     EV<<"Started, waiting for DYNA_CONN_REQ\n"
    Line 19:     pk = (DynaPacket *)receive();
    Line 20:     clientAddr = pk->getSrcAddress();
    Line 21:     ownAddr = pk->getDestAddress();
    // To set the module name to something informative
    Line 22:     char buf[30];
    Line 23:     sprintf(buf, "serverproc%d-clientaddr%d",
        getId(),clientAddr);
    Line 24:     setName(buf);
    // To respond to CONN REQ by CONN ACK
    Line 25:     EV << "client addr" << clientAddr << ", sending DYNA_CONN_ACK\n"
    Line 26:     pk->setName("DYNA_CONN_ACK");
    // DYNA_CONN_ACK kind is declared within an enum
    Line 27:     pk->setKind(DYNA_CONN_ACK);
    Line 28:     pk->setSrcAddress(ownAddr);
    Line 29:     pk->setDestAddress(clientAddr);
    Line 30:     pk->setServerProcId(getId());
    Line 31:     sendDirect(pk, serverOutGate);
    // To process data packets until DISC_REQ comes
    Line 32:     for ( ; ; ) {
    Line 33:         EV << "waiting for DATA(query) (or DYNA_DISC_REQ)\n"
    Line 34:         pk = (DynaPacket *)receive();
    Line 35:         int type = pk->getKind();
    Line 36:         if (type == DYNA_DATA)
    Line 37:             break;
    Line 38:         if (type !== DYNA_DATA)
    Line 39:             throw cRuntimeError("protocol error!");
    Line 40:         datapk = (DynaDataPacket *)pk;
Listing 11.11 describes the implementation of Switch.ned and it simulates a switch between clients and server.

Listing 11.11 Switch.cc

```plaintext
#include "DynaPacket_m.h"
using namespace omnetpp;
#define STACKSIZE 16384

class Switch : public cSimpleModule{
    public:
    Switch() : cSimpleModule(STACKSIZE){ }
    virtual void activity() override;
};

Define_Module(Switch);

void Switch::activity(){
    simtime_t pkDelay = 1 / (double)par("pkRate");
    int queueMaxLen = (int)par("queueMaxLen");
    // Queue creation at switch
    cQueue queue("queue");
    for ( ; ; ) {
        // To receive message
        cMessage *msg;
        if (!queue.isEmpty())
            msg = (cMessage *)queue.pop();
        else
            msg = receive();
```
// To model processing delay; packets that arrive meanwhile are queued
Line 20: waitAndEnqueue(pkDelay, &queue);
// To send message to destination
Line 21: DynaPacket *pk = check_and_cast<DynaPacket *> (msg);
Line 22: int dest = pk->getDestAddress();
Line 23: EV << "Relaying msg to addr=" << dest << "\n";
Line 24: send(msg, "port$o", dest);
/*
 Icon color changing to display the communication status
 normal=queue empty, yellow=queued packets; red=queue overflow
*/
Line 25: int qLen = queue.getLength();
Line 26: if (hasGUI())
Line 27: getDisplayString().setTagArg("i", 1, qLen == 0 ? "" : qLen<queueMaxLen ? "gold" : "red");
// To model finite queue size
Line 28: while (queue.getLength()>queueMaxLen) {
Line 29: EV << "Buffer overflow, discarding "
Line 30: << queue.front()->getName() << endl;
Line 31: delete queue.pop();
Line 32: }
Line 33: }
Listing 11.12 describes the implementation of Client.ned.

Listing 11.12 Client.cc

Line 1: #include "DynaPacket_m.h"
Line 2: using namespace omnetpp;
Line 3: #define STACKSIZE 16384

Line 4: class Client : public cSimpleModule{
Line 5: public:
Line 6: Client() : cSimpleModule(STACKSIZE) {} }
Line 7: virtual void activity() override;
Line 8: }

Line 9: Define_Module(Client);

Line 10: void Client::activity(){
// To query module parameters
Line 11: simtime_t timeout = par("timeout");
Line 12: cPar& connectionIaTime = par("connIaTime");
Line 13: cPar& queryIaTime = par("queryIaTime");
Line 14: cPar& numQuery = par("numQuery");
Line 16: DynaDataPacket *query, *answer;
Line 17: int actNumQuery = 0, i = 0;
Line 18: WATCH(actNumQuery);
Line 19: WATCH(i);
// To assign address: index of Switch's gate to which we are connected
Line 20: int ownAddr = gate("port$o")->getNextGate()->getIndex();
Line 21: int serverAddr = gate("port$o")->getNextGate()->size()-1;
Line 22: int serverprocId = 0;
Line 23: WATCH(ownAddr);
Line 24: WATCH(serverAddr);
WATCH(serverprocId);
for( ; ; ) {
    if(hasGUI())
        getDisplayString().setTagArg("i", 1, "\n");
// To keep an interval between subsequent connections
    wait((double)connectionIaTime);
    if(hasGUI())
        getDisplayString().setTagArg("i", 1, "green");
// For connection setup
    EV << "sending DYNA_CONN_REQ\n"
    connReq = new DynaPacket("DYNA_CONN_REQ", DYNA_CONN_REQ);
    connReq->setSrcAddress(ownAddr);
    connReq->setDestAddress(serverAddr);
    send(connReq, "port$o");
    EV << "waiting for DYNA_CONN_ACK\n"
    connAck = (DynaPacket *)receive(timeout);
    if(connAck == nullptr)
        goto broken;
    serverprocId = connAck->getServerProcId();
    EV << "got DYNA_CONN_ACK, my server process is ID="
    <<serverprocId << endl;
    delete connAck;
    if(hasGUI()) {
        getDisplayString().setTagArg("i", 1, "gold");
        bubble("Connected!");
    }
// For communication
    actNumQuery = (long)numQuery;
    for(i = 0; i < actNumQuery; i++) {
        EV << "sending DATA(query)\n"
        query = new DynaDataPacket("DATA(query)", DYNA_DATA);
        query->setSrcAddress(ownAddr);
        query->setDestAddress(serverAddr);
        query->setServerProcId(serverprocId);
        query->setPayload("query");
        send(query, "port$o");
        EV << "waiting for DATA(result)\n"
        answer = (DynaDataPacket *)receive(timeout);
        if(answer == nullptr)
            goto broken;
        EV << "got DATA(result)\n"
        delete answer;
        wait((double)queryIaTime);
    }
    if(hasGUI())
        getDisplayString().setTagArg("i", 1, "blue");
// For connection teardown
    EV << "sending DYNA_DISC_REQ\n"
    discReq = new DynaPacket("DYNA_DISC_REQ", DYNA_DISC_REQ);
    discReq->setSrcAddress(ownAddr);
    discReq->setDestAddress(serverAddr);
    discReq->setServerProcId(serverprocId);
    send(discReq, "port$o");
    EV << "waiting for DYNA_DISC_ACK\n"
    discAck = (DynaPacket *)receive(timeout);
    if(discAck == nullptr)
        goto broken;
Line 77: EV << "got DYNA_DISC_ACK\n";
Line 78: delete discAck;
Line 79: if(hasGUI())
Line 80: bubble("Disconnected!");
Line 81: continue;
//For error handling
Line 82: broken:
Line 83: EV << "Timeout, connection broken!\n";
Line 84: if(hasGUI())
Line 85: bubble("Connection broken!");
Line 86: }
Line 87: }

Listing 11.13 shows real-time scheduler for configuration.

<table>
<thead>
<tr>
<th>Listing 11.13</th>
<th>realtime.ini</th>
</tr>
</thead>
<tbody>
<tr>
<td># To enable real-time mode</td>
<td></td>
</tr>
<tr>
<td># realtimescheduler-scaling option allows scaling simulation execution</td>
<td></td>
</tr>
<tr>
<td># Default cSequentailScheduler event scheduler class</td>
<td></td>
</tr>
<tr>
<td># Here, scheduler-class is “cRealTimeScheduler”</td>
<td></td>
</tr>
<tr>
<td># cRealTimeScheduler represents ratio of simulation time to real-time</td>
<td></td>
</tr>
<tr>
<td># realtimescheduler-scaling = 2 or 0.5 cause simulation time to</td>
<td></td>
</tr>
<tr>
<td># progress twice or half as fast as real-time.</td>
<td></td>
</tr>
<tr>
<td>Line 1: [General]</td>
<td></td>
</tr>
<tr>
<td>Line 2: scheduler-class = &quot;cRealTimeScheduler&quot;</td>
<td></td>
</tr>
<tr>
<td>Line 3: realtimescheduler-scaling=0.5</td>
<td></td>
</tr>
<tr>
<td>#omnetpp.ini configuration should be part of realtime.ini</td>
<td></td>
</tr>
<tr>
<td>Line 4: include omnetpp.ini</td>
<td></td>
</tr>
</tbody>
</table>

11.4 Step-by-Step Execution

At the start of the simulation, the Tekenv GUI will appear that supports interactive simulation execution. Figure 11.4 shows simulation execution toolbar of OMNeT++ Tekenv GUI. Tekenv GUI toolbar contains a button labeled “STEP”, which is used to step-through simulation execution. The “RUN” button shown in Figure 11.4 is used to run the simulation as a whole with full animation. The “FAST” execution option used to run the simulation at faster execution speed with no animation and minimal debugging support (inspector updates). The “EXPRESS” execution option used to run the simulation at full execution speed with no animation and debugging support. The “RUN UNTIL” button is used to run the simulation until simulation breakpoint is reached, a time event occurs, or a certain predefined message is spotted in the code.

Figure 11.4: Simulation execution toolbar

At simulation load, module log of Tekenv interactive GUI shows that ClientServer network has been initialized with all submodules (i.e., Client, Server, Switch) as an initial step (stage 0). In addition, initialization is also done for the server, switch, and client’s associated ports/channels, as shown in log snippet 11.1.
**Initialization network and module phase**

**Step 1:** Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 1:**

Modules with activity() function need messages known as *starter* messages to boot. The OMNeT++ simulation kernel is responsible for inserting starter messages in the Future Event Set (FES) automatically for modules with activity module. At time $t = 0$, two events are shown in log snippet 11.2. Both the events represent the enqueuing of starter messages for switch/client modules.

**Log Snippet 11.2:** Starter messages at Switch and Client module

**Step 2:** Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 2:**

The output of step 2 is shown in log snippet 11.3. Here, client module (at timeout event of self-message) sends a connection request (as shown by the red dot in Figure 11.5) followed by waiting for an acknowledgment from the server side. This connection request will be received by switch module at the same time.

**Log Snippet 11.3:** Sending connection request

**Step 3:** Click STEP button (shown in Figure 11.4) three times.

**Outcome of Step 3:**

The output of step 3 is shown in log snippet 11.4. Here, on self-message timeout, switch module will relay received a connection request to the server module (as shown in Figure 11.6). Upon reception, at initialization, server process will be created (having ID 9) and started. The fundamental purpose of this server process module is to handle incoming connection requests from client side (waiting for connection request) shown in log snippet 11.4.

**Log Snippet 11.4:** Creation and initialization of server process
Step 4: Click STEP button (shown in Figure 11.4) one time.

Outcome of Step 4:

The output of step 4 is shown in log snippet 11.5. The server sends an acknowledgement to switch module (as shown in Figure 11.7) and server process for current client moves into the next state of waiting for DATA (query) requests.

** Event #5 t=6.645411712125 ClientServer.server.serverproc (ServerProcess, id=9) on DYNA_CONN_REQ (Dynapacket, id=2)
INFO [ServerProcess]ClientServer.server.serverproc9-clientaddr0: client is addr0, sending DYNA_CONN_ACK
INFO [ServerProcess]ClientServer.server.serverproc9-clientaddr0: waiting for DATA (query) (or DYNA_DISC_REQ)

Log Snippet 11.5: Acknowledgement for connection request

Figure 11.5: Connection request from Client

Figure 11.6: Relaying connection request

Figure 11.7: Connection ACK from Server

Figure 11.8: Relaying of connection ACK to Client

Step 5: Click STEP button (shown in Figure 11.4) two times.
Outcome of Step 5:

Output of step 5 is shown in log snippet 11.6. The switch receives connection acknowledgment packet from server and upon self-starter message timeout relays acknowledgment packet to the client module as shown in Figure 11.8.

```
** Event #9  t=8.655411712325  ClientServer.switch (Switch, id=3)  on DYN_CONN_ACK (DynaPacket, id=2)
** Event #10 t=9.022070378922  ClientServer.switch (Switch, id=3)  on selfmsg timeout-3 (omnetpp::cMessage, id=0)
INFO (Switch)ClientServer.switch: Relaying msg to addr=0
```

Log Snippet 11.6: Relaying of ACK towards Client

Step 6: Click STEP button (shown in Figure 11.4) one time.

Outcome of Step 6:

The output of step 6 is shown in log snippet 11.7. On receiving connection acknowledgment, client module immediately sends a query message to the server process (with ID 9) and moves into waiting for state for its results as shown in Figure 11.9.

```
** Event #11 t=9.382078538592  ClientServer.client[0] (Client, id=4)  on DYN_CONN_ACK (DynaPacket, id=2)
INFO (Client)ClientServer.client[0]: gct DYN_CONN_ACK, my server process is ID=9
INFO (Client)ClientServer.client[0]: sending DATA(query)
INFO (Client)ClientServer.client[0]: waiting for DATA(result)
```

Log Snippet 11.7: Sending query on ACK reception

Figure 11.9: Sending of data (query) to Server

Step 7: Click STEP button (shown in Figure 11.4) two times.

Outcome of Step 7:

The output of step 7 is shown in log snippet 11.8. The switch receives a data packet from a client and on self-starter message timeout, relays this packet towards the server module (shown in Figure 11.10).

```
** Event #12 t=9.3420761928992  ClientServer.switch (Switch, id=3)  on DATA(query) (DynaDataPacket, id=9)
** Event #13 t=10.006745946669  ClientServer.switch (Switch, id=3)  on selfmsg timeout-3 (omnetpp::cMessage, id=0)
INFO (Switch)ClientServer.switch: Relaying msg to addr=1
```
**Log Snippet 11.8:** Relaying query to Server

**Step 8:** Click STEP button (shown in Figure 11.4) one time.

**Outcome of Step 8:**

The output of step 8 is shown in log snippet 11.9. On receiving a data packet, server module redirects this query message to server process with ID = 9 (which was initiated at step 3). Server process with ID 9 is responsible for replying this query results to the current client.

```
** Event #14 t=10.018745045599 ClientServer.server (Server, id=2) on DATA(query) (DynaDataPacket, id=9)
INFO (Server)ClientServer.server: Redirecting msg to process ID=9
```

**Log Snippet 11.9:** Query redirection to server process

**Step 9:** Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 9:**

The output of step 9 is shown in log snippet 11.10. Processing will be done on received query and results will be sent back to the client (shown in Figure 11.11). Server process for this particular request will now move in the state of waiting for disconnection request from the client.

```
** Event #15 t=10.018745045599 ClientServer.server.serverproc9-clientDefault (ServerProcess, id=9) on DATA(query) (DynaDataPacket, id=9)
INFO (ServerProcess)ClientServer.server.serverproc9-clientDefault: got DATA(query), processing...
** Event #16 t=10.215940456599 ClientServer.server.serverproc9-clientDefault (ServerProcess, id=9) on selfmsg timeout-9 (onhttp::selfmsg, id=9)
INFO (ServerProcess)ClientServer.server.serverproc9-clientDefault: sending DATA(result)
INFO (ServerProcess)ClientServer.server.serverproc9-clientDefault: waiting for DATA(query) (on DATA_RECV_MSG)
```

**Log Snippet 11.10:** Processing and reply of query result

![Figure 11.10: Relaying of data to Server](image1)

![Figure 11.11: Sending query result to Client](image2)

**Step 10:** Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 10:**

The output of step 10 is shown in log snippet 11.11. On self-message timeout event, switch forwards results towards the client module (shown in Figure 11.12).

```
** Event #17 t=10.228748448899 ClientServer.switch (Switch, id=8) on DATA(result) (DynaDataPacket, id=9)
** Event #18 t=10.2956125128326 ClientServer.switch (Switch, id=8) on selfmsg timeout-3 (onhttp::selfmsg, id=0)
INFO (Switch)ClientServer.switch: Relaying msg to addr=0
```

155
Log Snippet 11.11: Query result forwarding to Client

Step 11: Click STEP button (shown in Figure 11.4) two times.

Outcome of Step 11:

The client receives a packet from the server and may send next query for the same set of processing (described in steps 6 to 11) as shown in log snippet 11.12.

Log snippet 11.12: Next query sending

The client may send disconnection request (shown in Figure 11.13) to server process as shown in log snippet 11.13.

Log snippet 11.13: Sending disconnection request
**Step 12**: Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 12**:

The output of step 12 is shown in log snippet 11.14. The switch receives and relays disconnection request to the server process through server module as shown in Figure 11.14.

```
** Event #10  t=18.054599732781  ClientServer.switch (Switch, id=3) on DYNA_DISC_REQ (DynaPacket, id=21)
** Event #11  t=18.721663399418  ClientServer.switch (Switch, id=3) on sel.fmag timeout-3 (cmnetpp::Message, id=0)
INFO (Switch)ClientServer.switch: Relaying msg to addr=1
```

**Log snippet 11.14**: Relaying of disconnection request to Server

**Step 13**: Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 13**:

The output of step 13 is shown in log snippet 11.15. Server redirects disconnection request to the respective server process (with ID = 9). Server process replies disconnection acknowledgment (shown in Figure 11.15) and closes this server process ID.

```
** Event #12  t=18.731066339418  ClientServer.server (Server, id=2) on DYNA_DISC_REQ (DynaPacket, id=21)
INFO (Server)ClientServer.server: Redirecting msg to process ID=9
** Event #13  t=18.731066339418  ClientServer.server.serverproc-clientAddr9 (ServerProcess, id=8) on DYNA_DISC_REQ (DynaPacket, id=21)
INFO (ServerProcess)ClientServer.server.serverproc-clientAddr9: got DYNA_DISC_REQ, sending DYNA_DISC_ACK
INFO (ServerProcess)ClientServer.server.serverproc-clientAddr9: killing
```

**Log snippet 11.15**: Sending disconnection ACK to Client

**Step 14**: Click STEP button (shown in Figure 11.4) two times.

**Outcome of Step 14**:

The output of step 14 is shown in log snippet 11.16. Switch on self-message timeout redirects disconnection request to the client module as shown in Figure 11.16.

```
** Event #14  t=18.741966339418  ClientServer.switch (Switch, id=3) on DYNA_DISC_ACK (DynaPacket, id=21)
** Event #15  t=18.407733566085  ClientServer.switch (Switch, id=3) on sel.fmag timeout-3 (cmnetpp::Message, id=0)
INFO (Switch)ClientServer.switch: Relaying msg to addr=0
```

**Log snippet 11.16**: Relaying disconnection ACK to Client

---

**Figure 11.15**: Disconnection ACK

**Figure 11.16**: Relaying of disconnection ACK
Step 15: Click STEP button (shown in Figure 11.4) two times.

Outcome of Step 15:

The output of step 15 is shown in log snippet 11.16. On receiving disconnection acknowledgment, Client module will be virtually disconnected from server module and for the new data (query) request, all steps (from step 2 to 16) are required to be followed.

```
** Event #36 t-19.417731066088 ClientServer.client[0] (Client, id=4) on DYNA_DISC_ACK (DynaPacket, id=21)
INFO (Client)ClientServer.client[0]: got DYNA_DISC_ACK
```

Log snippet 11.16: Disconnection ACK at Client

Practice Task

1. You are required to implement the practice tasks given in Chapter 7 (working of ARQ), Chapter 8 (working of Sliding Window), and Chapter 9 (multiplexing/de-multiplexing) through activity() function discussed in chapter 11.
REFERENCES

Reliable Transmission, 67
Report generator, 12
RINASim, 17
Satellite, 6
Satellites, 6
scheduleAt(), 58
scheduler-class, 143
Server, 5, 34, 110, 130, 131, 132, 134, 135, 137
setTagArg(), 109, 120, 121, 122, 123
Simple Module, 19, 20, 28
sim-time-limit, 21, 133
simtime-resolution, 21
Simulation model, 9, 151
Simulation Models, 9
SimuLTE, 17
Sliding Window, 67, 78, 79, 80, 81, 87
snapshot-file, 133
Stop-and-Wait, 63, 67, 68, 79
Switches, 2
Taxonomy, 4, 6
Tekenv, 33, 54, 66, 108, 109, 143
Terrestrial, 6
Time-driven, 11
Topology, 19, 22, 27, 34, 55, 66, 78, 91
total-stack, 133
Trace-driven, 10
Veins, 17
volatile, 135
WAN, 6, 7
WATCH, 109, 127, 128, 138, 141
WBAN, 6, 7
Wide Area Network, 6, 7
WBAN, 6, 7
Wireless, 5, 6, 8, 13, 16, 151, 152
Wireless Body Area Network, 6, 7
Wireless Local Area Network, 6, 7
Wireless Metropolitan Area Network, 6, 7
Wireless Personal Area Network, 6, 7
Wireless Wide Area Network, 6, 7
WLAN, 6, 7
WMAN, 6, 7
WPAN, 6, 7
WWAN, 6, 7